6502® USER'S MANUAL

JOSEPH J. CARR

6502 User's Manual

Joseph J. Carr

6502 User's Manual

A Reston Computer Group Book
Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

Library of Congress Cataloging in Publication Data

Carr, Joseph J.
    6502 user's manual.

    "A Reston Computer Group book."
    1. 6502 (Microprocessor) I. Title. II. Title: Six
thousand five hundred two user's manual.
QA76.8.S63C37 1984 001.64 83-26887
ISBN 0-8359-7004-6
ISBN 0-8359-7002-7 (pbk.)

10 9 8 7 6 5 4 3 2 1

PRINTED IN THE UNITED STATES OF AMERICA

Contents

Introduction

This book is intended to be both an instruction guide and a tool . . . but mostly a utilitarian tool for the programmer/interfacer of 6502-based microcomputers. It is intended that this book become dog-eared and worn from heavy use on a programmer's table or a hacker's workbench . . . use it and abuse it, that's why I wrote it. There is no attempt in this book to show you how much the author knows about computers—you don't have time for trash like that. The information in this book was selected for its usefulness to a wide variety of readers. Like its earlier companion volume, *The Z80 User's Manual* (Joseph J. Carr, Reston Publishing Company), the book is intended to collect into one place most, perhaps all, the information you need for assembly and machine language programming and for performing hardware interfacing chores. Of course, with all of the different kinds of 6502-based machines on the market today, it was impossible to provide details of all machines, so I stuck pretty much to generic details applicable to any 6502-based system.

New features in this book that are not in my earlier Z80 book include a discussion of the different types of computers. My earlier assumption (which was then almost true!) was that the reader already possessed a basic knowledge of computers and at least BASIC programming. But today, the microcomputer biz has doubled, tripled, fourpled, and fippled in just a little while and there are now many thousands in the market who do not necessarily possess the semi-advanced background of the former readership. Back when the *Altair* S-100 was king, and the "Great Woz" was taking his first byte out of the *Apple,* anybody who wanted in on these microcomputer widgets was, by definition, a genuine hacker. The market was self-defining to an extent, because only those who would become hackers *dared* (that's right, *dared!*) buy a *computer kit* (*shudder* . . . the complexity of it all). But today, with prices down and mass marketing technique up, everybody can own a computer and do more with it than we late-sixties engineering students could do with three-and-a-quarter tons of junk! Being a semi-irreverent sort, I aim to tear off the robes of the modern sacerdotal priests of "HIGH TECHNOLOGY" so that one and all can enjoy and benefit from the marvelous little microcomputer. Toward this end, Chapter One of this book contains several features. One is a discussion of microcomputers as opposed to minicomputers, "mainframe" computers, and so forth. Those descriptions will make you conversant with the language of the technology even if you are scared to death to "hit RETURN" on your new toy! We will also look at some applications categories for microcomputers, and some of the more popular 6502-based machines.

One feature most asked for by readers of my other books is a discussion on the basic operation of any programmable digital computer. It seems that most authors (myself included) have, in the past, launched right into the discussion of chips and bytes and other such wonders without ever stopping to ask the poor confused beginner whether or not he or she actually knows how a programmable digital computer functions. *What does go on inside that box?* Toward that end, I have created a hypothetical computer that is not based on any microprocessor chip (that way I can use it in several more books!), but it is based on the generic form of the computer. I call this device the *Mythical Analytical Device,* or MAD (which acronym, by the way, accurately describes the typical user after the umpteenth "bomb" and the apparent state of us elderly hackers—as seen by computer newcomers).

The book also contains a lot of material on interfacing to the 6502 microprocessor, and to computers that are based on the 6502. I have also covered some of this material in my book *Elements of Microcomputer Interfacing,* also a Reston book. The material in that book is more general in nature, but goes a lot deeper than was possible within the length/cost constraints placed on the present work.

The 1980 companion volume to this book is *Z80 User's Manual.* That book was rejected by two other publishers before Reston took a chance and published the work. That risk was apparently well taken, however, because the *Z80 User's Manual* turned out to be a popular best-seller and was even recommended in the Timex/Sinclair T/S-1000 *User's Manual.* The success of that book startled me (and at least one of the editors who rejected it), and I could never figure out why anyone would buy it. Most of the material given in that book, and in large sections of this book, is available elsewhere for free. So why, I wondered, would anyone pay me good money for the information that is in so many other sources? One reader of mine (who is also a friend and fellow hacker) gave me the answer: that book (and this one) sums up, explains, interprets, and collects into one volume instead of ten what everyone needs to know about their machines. Rather than a stack of books, you only need two: this book and the operating manual for your computer! (well, you could buy a few more of my books if you insist!)

Speaking of buying books. If you are just browsing right now, *go ahead and buy the book* (please don't shoplift it . . . that's naughty!), I need the money!

Joe Carr

1

Introduction to Microprocessors and Microcomputers

One of the most frequent questions the microcomputer owner asks is, "What will it do?" This question is exasperating because it has too many answers. Indeed, what is the role of a microcomputer? For that matter, what is a microcomputer?

At one time, definitions were simpler. As a freshman engineering student, I was allowed to use an IBM® 1601-1620 machine; *that* was a computer! There was no doubt in anyone's mind about that machine's identity; it took up an entire room on the second floor of the engineering school's building. But, today, an engineering student can sit at a small desk with an Apple® II (complete with video CRT display, printer, and two disc drives) that has more computing power than that old 1601! In fact, many engineering students find the cost of the typical small system so affordable that they can own their own computer. Now the student can have more computing power in a dorm room than we had in the school of engineering. The cost of the modern microcomputer is less than one-tenth of what one of the lesser machines cost only a decade ago—not counting the fact that 1971 dollars were bigger than today's dollars.

Before attempting to define the role of the microcomputer, let's first try to define what the microcomputer is. Terminology in the computer field is often "O.B.E."—overcome by events. For example, consider the terms *microcomputer* and *minicomputer*. Some of us use these terms interchangeably, because modern single-chip computers (e.g., the Intel 8048) tend to make such usage seem reasonable. But, for our purposes, we require sharply focused meanings for these two terms and others: *minicomputer, microcomputer, single-chip computer, single-board computer,* and *mainframe computer.*

MICROPROCESSORS

The microprocessor is a large scale integration (LSI) integrated circuit (IC) that contains the central processing unit (CPU) of a programmable digital computer. The CPU of a computer contains the arithmetic logic unit (ALU) that performs the basic computational and logical operations of the computer. The CPU also houses the control logic section, which performs housekeeping functions, and may or may not have several registers for the temporary storage of data. All CPUs have at least one temporary storage register called the accumulator, or A-register. The principal attribute of a microprocessor is that it will execute instructions sequentially. These instructions are stored in coded binary form in an external memory.

MICROCOMPUTERS

A microcomputer is a full-fledged programmable digital computer that it built around a miçroprocessor "chip," i.e., integrated circuit; the microprocessor acts as the CPU for the computer. In addition to the microprocessor chip, the microcomputer typically will have additional chips; the number may vary from two to hundreds depending upon the design and the application. These external chips may provide such functions as memory (both temporary and permanent), input/output (I/O), and other functions. The microcomputer may be as simple as a KIM-1, or as complex as a 30-board professional machine with all the electronic data processing "goodies."

SINGLE-CHIP COMPUTERS

For several years we had no excuse for interchanging the terms *microprocessor* and *microcomputer;* a μP was an LSI chip and μC was a computing machine. But with the advent of the 8048 and similar devices, previously well-defined boundaries dissolved because these devices were both an LSI IC *and* a computer. A typical single-chip computer may have a CPU section, two types of internal memory (temporary and long-term permanent storage), and at least two I/O ports. Some machines are even more complex.

The single-chip computer does, however, require some external components before it can do work. By definition, the microcomputer already has at least a minimum of components needed to perform a job.

SINGLE-BOARD COMPUTERS

The single-board computer is a programmable digital computer, complete with input and output peripherals, on a single printed circuit board. Popular 6502-based examples are the KIM-1, SYM-1, and AIM®-65 machines. The single-board computer might have either a microprocessor or a single-chip computer at its heart.

The peripherals on a single-board computer are usually of the most primitive kind (e.g., AIM-65), consisting of 7-segment LED numerical displays and hexadecimal keypads reminiscent of those on a handheld calculator or *Touch-Tone*™ telephone. The typical display is capable of displaying only hexadecimal numeral characters because of the form constraints of using 7-segment LED display devices. The Rockwell International AIM-65 uses a regular ASCII keyboard and a 20-character display made of 5 × 7 dot matrix LEDs. In addition, the AIM-65 has a built-in 20-column dot matrix thermal printer that uses printing calculator paper.

Most single-board computers have at least one interface connector that allows either expansion of the computer or interfacing into a system or instrument design.

The manufacturers of SBCs, such as the KIM-1 and others, probably did not envision their wide application as a small-scale development system. These computers were primarily touted as trainers for use in teaching microcomputer technology. But for simple projects such computers also work well as a mini-development system! More than a few SBC trainers have been used to develop a microcomputer-based product, only to wind up being specified as a "component" in the production version. In still other cases, the commercially available SBC has been used as a component in prototype systems, and then, in the production version, a special SBC (lower cost) was either bought or built.

MINICOMPUTERS

The minicomputer predates the microcomputer and was originally little more than a scaled-down version of larger data processing machines. The Digital Equipment Corporation (DEC®) PDP-8 and PDP-11™ machines are examples of "minis." The minicomputer will use a variety of small-scale (SSI), medium-scale (MSI), and large-scale integration (LSI) chips.

Minicomputers have traditionally been more powerful than microcomputers. They had, for example, longer length binary data words (12 to 32 bits instead of 4 or 8 bits found in micros), and operated at faster speeds (6 to 12 mHz instead of 1 to 3 mHz). But in this area, distinctions are fading. Digital Equipment Corporation, for example, offers the LSI-11 microcomputer that acts like a mini. Similarly, 16-bit microcomputers are available, as are 6 mHz devices. It is sometimes difficult to draw the line when a microcomputer is in the same size cabinet as a minicomputer, and minicomputers can be bought in desk-top configurations!

MAINFRAME COMPUTERS

The large computer that comes to mind when most people think of computers is the mainframe computer. These machines are the computers used in large-scale data processing departments. Microcomputerists who have an elitist mentality sometimes call mainframe computers "dinosaurs." But, unlike their reptilian namesakes, these dinosaurs show no signs of extinction and are, in fact, an evolving species. The IBM 370 is an example of a mainframe computer.

ADVANTAGES OF MICROCOMPUTERS

Microcomputers have certain advantages, as attested to by the fact that so many are sold! But what are these advantages?

The most obvious advantage of the microcomputer is reduced size; compared with dinosaurs, microcomputers are mere lizards! An 8 bit microcomputer with 64K bytes of memory can easily fit inside a table-top cabinet. For example, Apple® III (Figure 1-1) fits the complete computer (plus one optional disc drive) into the space inside a small table-top cabinet! Another company packs a computer with 16K of random access memory (RAM) inside a keyboard housing!

The LSI microcomputer chip is generally more complex than a discrete components circuit that does the same job. However, the interconnections between circuit elements are much shorter (micrometers instead of millimeters). Input capacitances are thereby made lower. The MOS technology used in most of these ICs produces very low current drain, hence the overall reduced heating. While a minicomputer may require a pair of 100 cfm blowers to keep the temperature within specifications, a microcomputer may be able to use a single 40 cfm muffin fan or no fan at all!

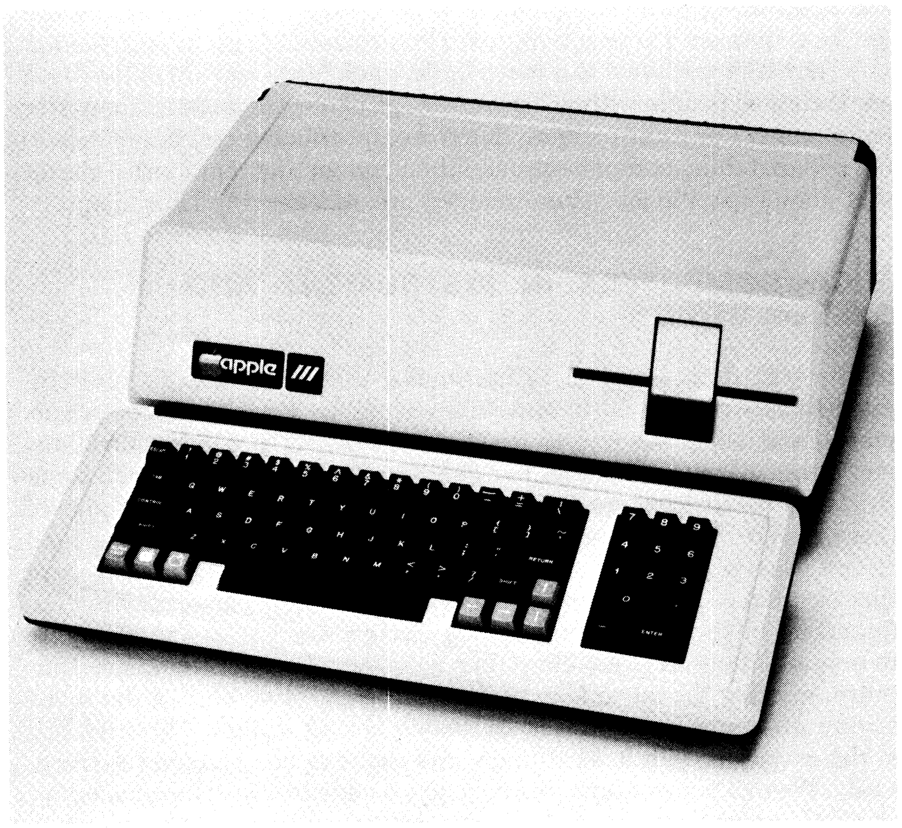


Figure 1-1. Apple III microcomputer

Another advantage of the LSI circuit is reduced component count, which reduces size. In addition it also affects reliability. If the LSI IC is just as reliable as any other IC (and so it seems), then the overall reliability of the circuit is increased dramatically. Even if the chip reliability is lower than in lesser ICs, we would still achieve superior reliability due to fewer interconnections on the printed circuit board, especially if IC sockets are used. Some of the most invidious troubleshooting problems result from defective IC sockets!

MICROCOMPUTER INTERFACING

The design of any device or system in which a microcomputer or microprocessor is used is the art of *defining* the operation of the system or device, *selecting the components* for the device or system, *matching* and *integrating* those components (if necessary), and *constructing* the device or system. These activities are known collectively as *interfacing.*

But let's get down to a more basic level. Most readers of this book are technical people with some knowledge of electronics and computer technology. For most readers, therefore, interfacing consists of selecting and matching components, and then connecting them into a circuit that does a specific job. These matters are addressed in later chapters.

MICROCOMPUTERS IN INSTRUMENT AND SYSTEM DESIGN

In the past designers had to use analog electronic circuits, electromechanical relays (which sometimes leads to a maintenance nightmare), and other devices to design instruments, process controllers, etc. These circuit techniques had their limitations and produced some irritating results; factors like thermal drift loomed large in some of these circuits. In addition, the design was cast in cement once the final circuit was worked out. Frequently, relatively subtle changes in a specification or requirement produced astonishing changes in the configuration of the instrument; analog circuits are not easily adaptable to new situations in many cases. But with the advent of the microcomputer, we gain the advantage of flexibility and solve some of the more vexing problems encountered in analog circuit design. The memory of the computer tells it what to do, and that can be changed relatively easily. We can, for example, store program code in a *read only memory,* or ROM, which is an integrated circuit memory. If a change is needed, then the software can be modified and a new ROM installed. If the microcomputer was configured well, then it is possible to redesign only certain interface cards (or none at all) to make a new system configuration! For example, an engineer built an anode heat computer for medical X-ray machines. A microprocessor would compute the heating of the anode as the X-ray tube operated, and would sound a warning if the limit of safety was exceeded—thus saving the hospital the cost of a $10,000 X-ray tube! But different X-ray machines require different interfacing techniques, a problem that previously had meant a new circuit design for each machine. But by intelligent engineering, the anode heat computer could be built with a single interface card that married the "universal" portion of the instrument with each brand of X-ray machine. Thus, the company could configure the instrument uniquely for all customers at a minimal cost.

Another instrument that demonstrates the universality of the microcomputer is a certain cardiac output computer. This medical device is used by intensive care physicians to determine the blood pumping capability of the heart in liters per minute. A "bolus" of iced or room tempterature saline solution is injected into the patient at the "input" end of the right side of the heart (the heart contains two pumps, right side and left side, with the right-side output feeding the left-side input via the lungs). The temperature at the output end of the right side is monitored, and the time integral of temperature determined. This integral, together with some constants, is massaged by the computer to calculate the cardiac output.

These machines come in two versions, research and clinical. The researcher will take time to enter certain constants that depend upon the catheter used to inject saline, temperature, and other factors, and will be more vigorous in following the correct procedure. But in the clinical setting, technique suffers as the patient is cared for, resulting in "machine error," which is actually operator error. To combat this problem, the manufacturer offers two machines. One is a research instrument and is equipped with front panel controls that allow the operator to select a wide range of options. The other, a clinical model, allows no options to the operator and is a "plug and chug" model. The interesting thing about these instruments is that they are *identical* on the inside! The only difference is the front panel and the position of an on-board switch! The manufacturer's program initially interrogates a switch to see if it is open or closed. If it is open, then it "reads" the keyboard to obtain the constants. On the other hand, if it is closed, then the program branches to a subprogram that assumes certain predetermined constants that are loaded on the buyer's prescription when the instrument is delivered. The cost savings of using a single design for both instruments are substantial!

6502-BASED MACHINES

We are going to examine some of the different 6502-based machines found on the market. Inclusion in here does not connote endorsement of the product, nor that another manufacturer's product isn't as good.

SYNERTEK SYM-1

Several years ago, the original manufacturer of the 6502 microprocessor, MOS Technology, Inc., produced a small, single–board computer that contained a hexadecimal keyboard and LED readouts. Originally conceived as a trainer, the KIM-1 microcomputer became something of a standard among single-board computers, and its "bus" is now sometimes referred to as the "KIM-bus." The KIM inspired a large collection of magazine articles, books, and accessory products. For many advanced computer scientists, the little KIM-1 was their first introduction into the world of microcomputer technology.

Although the SYM-1 microcomputer shown in Figure 1-2 is based on the original KIM-1 machine of another manufacturer, it extended the machine's capabilities and provides more features than the original design. Synertek Systems Corporation of Santa Clara, CA is the manufacturer of the SYM-1 machine.

Although the principal application for the SYM-1 is to train engineers and students in microcomputer interfacing and programming technology, applications have expanded into engineering laboratory work, prototyping of devices based on the 6502 microprocessor, instrumentation, and conducting both experimenting and testing in engineering and scientific laboratories. As the SYM-1 uses the same identical hardware interface bus as the earlier KIM-1 device, it may be "plugged into" applications previously reserved to the KIM-1 machine.

The SYM-1 device has a 4K byte on-board monitor program, 1K byte of on-board RAM (expandable to 4K bytes), and provision for up to 28K bytes of on-board ROM or PROM. The applications port has 15 bidirectional, TTL-compatible I/O lines, which, again, is expandable. The machine also offers data storage and program storage on

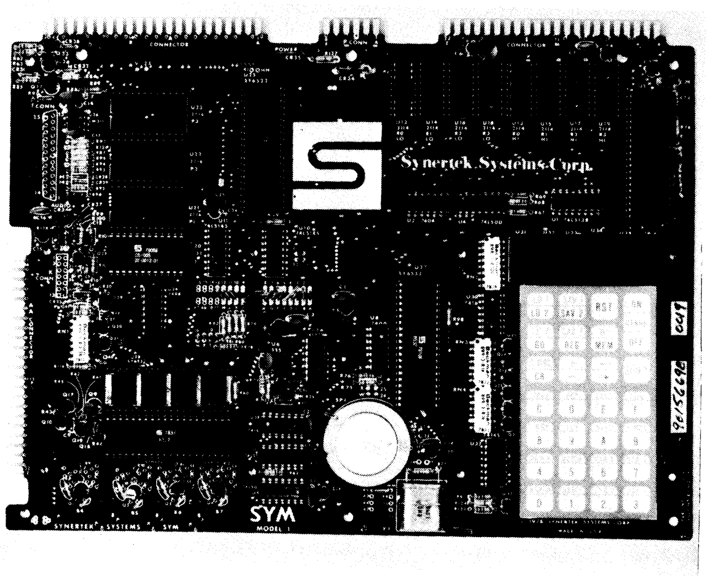


Figure 1-2. SYM-1 microcomputer

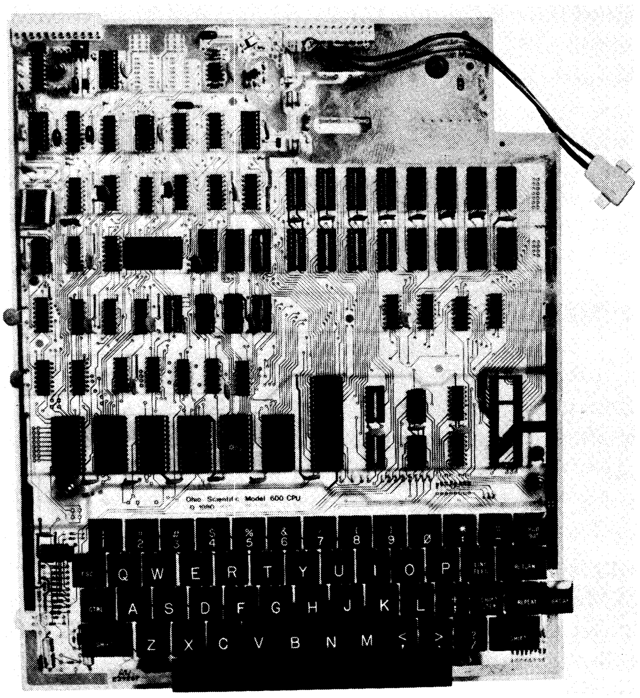


Figure 1-3. Ohio Scientific Superboard II

audio cassettes (an ordinary cassette tape player that has both "MIC" and "EAR" jacks can be used), and will accommodate a full duplex teletypewriter (TTY) 20 milliampere loop. This last feature makes the SYM-1 compatible, not just with TTY machines, but also with a wide variety of hard-copy printers now on the market. The machine includes one other I/O port, the common RS-232 serial interface port. The RS-232 port makes the SYM-1 compatible with a variety of video terminals and other peripherals. An on-board video terminal capability allows you to use either a TV monitor or, if an R.F. modulator is provided, a home TV receiver to receive output data (32 character line of video).

OHIO SCIENTIFIC SUPERBOARD II

The microcomputer in Figure 1-3 solves some of the problems inherent in other single-board designs such as inconvenient keyboard format. This machine also uses the same microprocessor (6502) as the KIM-1 and SYM-1 machines, although it does not use the KIM-1 bus. Programming and data entry are through a full ASCII keyboard like those found on video CRT terminals and larger computers.

The Superboard II can interface with TTY, CRT video terminals, and other peripherals. It is probably one of the simplest of the so-called "advanced" single-board computers and offers much that the lesser machines cannot, for example, more memory and programming in BASIC.

APPLE II AND III

The Apple II and its later cousin, the Apple III, shown in Figure 1-1, have become the byword in personal computers, partially because these computers make available a "full-service" microcomputer in a small package. A system that includes 48K bytes of memory, color TV graphics, color TV monitor, a teletypewriter, and two 5.25 inch disc drives can take up little more space than a table top.

The Apple II comes with a plug-in BASIC, with a more extensive version of BASIC available as an option. It also has an assembly language and built-in disassembler capability. The ordinary Apple II is available with an audio cassette interface, although for any serious work it is recommended that at least one disc be acquired.

Also built into the Apple II is a video display circuit that will drive an ordinary television monitor. The regular video format is 40 characters per line, with a total of 24 lines on the CRT screen at any one time. An interesting feature of the Apple II video monitor is that either regular (white characters on black background) or inverse (black characters on white background) modes can be used, and some of the characters can be programmed to flash on and off. The color graphics video display is capable of 15 different colors on a normal color video monitor.

A high-resolution video display provides 280h $\times$ 192v capability, allowing the programmer to draw graphs and other displays on the CRT screen.

MICROPROCESSOR FUNDAMENTALS

The microprocessor chip literally revolutionized the electronics industry. Although initially thought of as either a small logic controller or as a data processing machine (depending upon your perspective and the first chip you saw), the microprocessor blossomed in less than a decade into a major force with hundreds of applications.

What is a microprocessor? How does it relate to a microcomputer? We will explore these questions here, and hopefully present a good grounding in computer technology basics. But first, we will study computers in general by describing a "typical" programmable digital computer in block diagram form. In chapters to follow we will study the 6502 device.

Mythical Analytical Device (MAD)

Rather than mold our discussion around any one manufacturer's product, let's make up one that is general enough to cover a large number of actual devices. Our "computer" will be nicknamed the Mythical Analytical Device, or MAD, because the acronym MAD adequately describes both the emotional state of programmers (whose frustrations mount geometrically with each passing "bomb-out") and the mental health of computer sciences "freaks" (who are often seen wandering aimlessly through university corridors muttering the arcane glossolalian prayers of their modern religion, "Hail, microprocessor, from whom all bits and bytes emanate. . .").

Figure 1-4 shows the block diagram of MAD. Like any programable digital computer, MAD has three main parts: *central processing unit* (CPU), *memory*, and *input/output* (I/O). There are certainly other functions in some machines, but many are either special applications of these main groups or are too unique to be described in a general machine.

The central processing unit controls the operation of the entire computer. *Memory* can be viewed as an array of "cubbyholes," such as those used by postal workers (Figure 1-5) to sort mail. Each cubbyhole represents a specific address on the letter carrier's route. An address in the array can be uniquely specified (identifying only one location) by designating the row and column in which the cubbyhole is found. If we want to specify the memory location (i.e., cubbyhole) at row 3 and column 2, then we could create a row X column "address number" which, in this example, would be 32.

Each cubbyhole represents a unique location in which to store mail. In the computer, the memory location stores a binary word of information. In an 8-bit computer, each location will store a single 8-bit binary word. The different types of memory devices are discussed in another chapter.

The three lines of communication between the memory and the CPU are address bus, data bus, and control logic signals. These avenues of communication control the interaction between memory and I/O

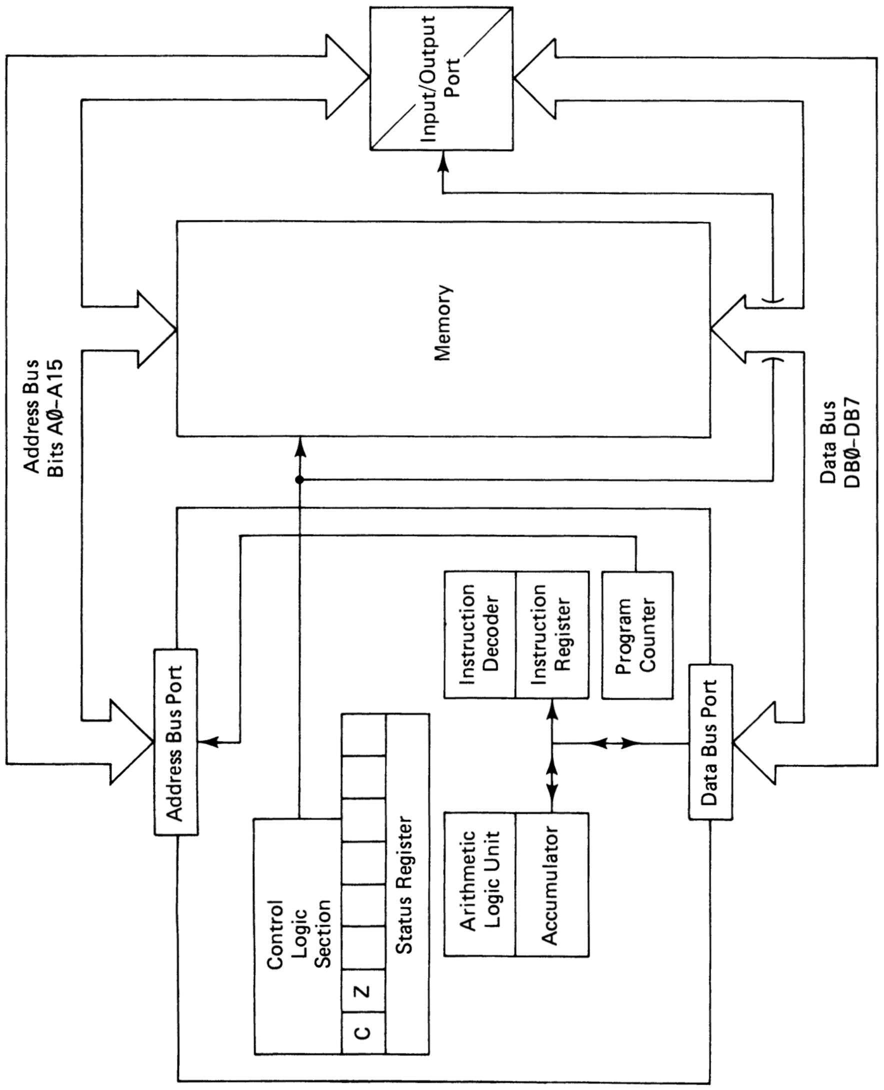


Figure 1-4. Block diagram for the mythical analytical device (MAD)

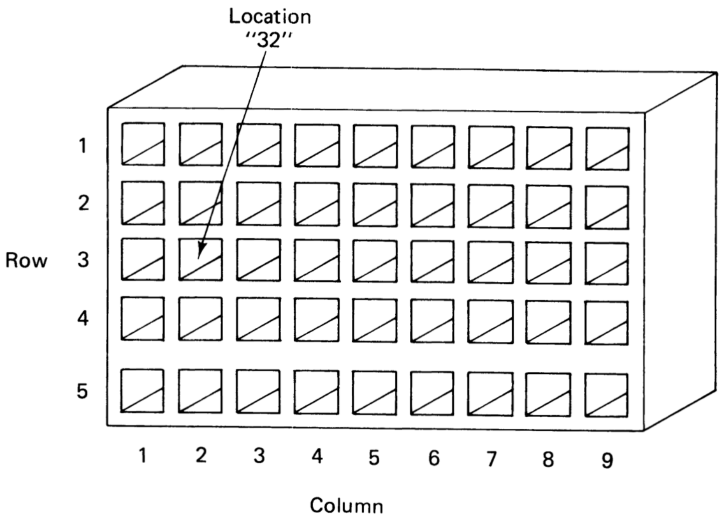


Figure 1-5. "Memory"

on the one hand, and the CPU on the other. Therefore they also control the functioning of the entire computer.

The address bus (bits A∅ through A15 in Figure 1-4) communicates to the memory bank the address of the exact memory location being called by the CPU, regardless of whether a read or write operation is taking place. The address bus consists of parallel data lines, one for each bit of the binary word that is used to specify the address location. In most 8-bit microcomputers, for example, the address bus consists of 16 bits. A 16-bit address bus can uniquely specify 2^{16}, or 65,536, different locations. This size is called "64K" not "65K" as one might expect. It seems that "k" represents the metric prefix *kilo,* which denotes 1,000. Since 2^{10} is 1,024, however, computer people long ago decided that kilo would be 1,024, not 1,000. The "big k" (1,024) is represented with an uppercase K rather than k, which is used for real kilo.

The size of memory which can be addressed doubles for every bit added to the length of the address bus. Hence, adding one bit to our 16-bit address bus creates a 17-bit address bus which can designate up to 128K locations. Some 8-bit machines which have 16-bit address buses can be made to look like bigger machines by certain tactics that make a longer pseudo-address bus. In those machines, several 64K memory banks are used to simulate continuously addressable 128K, 256K, or 512K memories.

The data bus is the communications channel over which data travels between the main register (called the accumulator or A-register) in the CPU and the memory. The data bus also carries data to and from the various input and/or output ports. If the CPU wants to "read" the data stored in a particular memory location, then that data is passed from the memory location over the data bus to the accumulator register in the CPU. Memory write operations are exactly the opposite direction, but otherwise the same.

The size of the data bus is usually cited as the "size" of the computer. Therefore, an 8-bit microprocessor/microcomputer is one that has an 8-bit data bus; a 16-bit microcomputer will have a 16-bit data bus. Do not be confused by salesmen such as the bozo who told me his 6502-based machine (8-bit data bus) was "in reality" a 16-bit machine because it had a 16-bit address bus!

The last memory signal is the control logic or timing signal. These are one or more binary logic signals that tell memory if it is being addressed, and whether the request is a read or write operation. The details of control logic signals differ between different microprocessor chips, so only those of the 6502 will be discussed in this book (for Z80 signals see *Z80 User's Manual*, by J. J. Carr, Reston Publishing Co.).

The input/output (I/O) section is the means by which the CPU communicates with the outside world. An input port will bring data in from the outside world and then pass it over the data bus to the CPU where it is stored in the accumulator. An output port reverses that data flow direction.

In some machines, separate I/O instructions are distinct from memory instructions. The Z80 is one such machine. The Z80 will pass the port address over the lower 8 bits of the 16-bit address bus (8-bit I/O address used in the Z80 can uniquely address up to 256 different ports). In other machines, such as the 6502, there are no distinct I/O instructions. In those machines, the I/O components are treated as memory locations; this technique is called *memory-mapping* or *memory-mapped I/O*. Input and output operations then become memory-read and memory-write operations, respectively.

Central Processing Unit (CPU)

The CPU is literally the heart and brains of any programmable digital computer, including MAD. Although there are some different "whistles and bells" features in certain machines, all will have the features shown in our MAD computer (Figure 1-4). The principal subsections of the CPU include (at least) the following: *accumulator* or *A-register, arithmetic logic unit* (ALU), *program counter* (PC), *instruction register, status register,* and *control logic section.*

The accumulator is the main register in the CPU, and will have the same bit length as the data bus. All instructions executed by the CPU involve data in the accumulator, unless otherwise specified in the description of that instruction. Therefore, an ADD instruction causes an arithmetic addition of the data cited by the instruction to the contents of the accumulator.

Although there are often other registers in the CPU, the accumulator is the main register. The main purpose of the accumulator is the temporary storage of data operated on by the instruction being executed. Data transfers to and from the accumulator are nondestructive. In other words, data "transfers" are not really transfers at all, but are, instead, "copying" operations. Suppose, for example, the hexadecimal number $8F_{16}$ is stored in the accumulator when an instruction is encountered requiring that the contents of the accumulator be stored at memory location $A008_{16}$. After the instruction is executed, we will find $8F_{16}$ both in memory location A008 *and* in the accumulator. If we have the opposite operation (i.e., transfer contents of accumulator to location $A008_{16}$), then we will see the same situation; after the transfer, the data will be in both locations. Since the accumulator contents change every time an instruction is executed, we will have to use such transfers to hold critical data some place in memory.

The arithmetic logic unit (ALU) contains the circuitry that performs the arithmetic operations of addition and (sometimes) subtraction, plus the logical operations of AND, OR, and XOR.

The program counter (PC) contains the address of the next instruction to be executed. The secret to the success of a programmable digital computer is its ability to fetch and execute instructions sequentially. Normally, the PC will increment appropriately (1, 2, 3, or 4) while executing each instruction: 1 for 1-byte instructions, 2 for 2-byte instructions, etc. For example, the instruction "LDA,n" is a 6502 instruction mnemonic that loads the accumulator with the number "n." In a program, we will find the code for LDA,n followed by "n."

Location	*Code*	*Mnemonic*
0100		LDA,n
0101	"n"	"n"
0102	(next instruction)	

At the beginning of this operation, PC = 0100, but after execution it will be PC = 0102 because LDA,n is a 2-byte instruction.

There are other ways to modify the program counter. For example, executing any form of JUMP instruction modifies the contents of PC to contain the address of the "jumped to" location. Another way to modify the PC contents is to activate the reset line. The computer sees reset as a hard-wired JUMP to location 0000.

The instruction register is the temporary storage location for instruction codes stored in memory. When the instruction is fetched from memory by the CPU, it will reside in the instruction register until the next instruction is fetched.

The instruction decoder is a logic circuit that reads the instruction register contents and then carries out the intended operation.

The control logic section takes care of housekeeping chores within the CPU, and issues or responds to control signals from the outside world. These signals are not universally defined (which is one reason why we will consider two chips later in this chapter), but control such functions as memory requests, I/O requests, read/write signaling, interrupts, etc.

The status register, also sometimes called status flags, is used to indicate the status of the CPU at any given instant to the program, and sometimes to the outside world. Each bit of the status register represents a different function. Different microprocessor chips use slightly different status register architectures, but all will have a carry flag (C) to indicate when an instruction execution caused a "carry," and a zero flag (Z) that indicates when an arithmetic or logic instruction resulted in zero or nonzero in the accumulator (typically, $Z = 1$ when the result is zero).

We have now developed the CPU for our MAD computer. This discussion in general terms also describes a typical microprocessor chip; a microprocessor (as opposed to a single-chip computer) is essentially the CPU portion of a MAD.

Operation of MAD

A programmable digital computer such as MAD operates by sequentially fetching, decoding, and then executing instructions stored in memory. These instructions are stored in the form of binary numbers. Some early machines had two memories, one each for program instructions and data. The modern method, however, uses the same memory for both data and instructions.

How does the dumb computer know whether the binary number stored in any particular location is an instruction, data, or an alphanumeric character representation (e.g., ASCII or Baudot codes)? The answer to this important question is the key to the operation of MAD: The MAD operates in cycles.

A computer will have at least two cycles: instruction fetch and execution, and in some machines these cycles are subcycles. The details differ even though general scenarios are similar.

Instructions are stored in memory as binary numbers called *operation codes*, or op-codes. During the instruction fetch cycle, an op-code will be retrieved from the memory location specified by the program counter and stuffed into the instruction register. The CPU assumes that the programmer was smart enough to arrange things such that an op-code will be stored at that location when the PC increments to that address.

During the first cycle, an instruction is fetched and stored in the instruction register. During the second cycle, the instruction decoder will read the IR, and then carry out the indicated operation. When these two cycles are completed, an instruction will have been fetched and executed, the program counter incremented to reflect the memory location that will contain the next instruction, and the CPU made ready for the next instruction. The CPU will then enter the next instruction fetch cycle and the process repeats itself.

This process continues over and over again as long as the MAD is working. Each step is synchronized by a train of clock pulses so that events remain rational.

This description illustrates what a computer can or cannot do. The CPU can shift data around, perform logical operations (e.g., AND, OR, XOR), add two N-bit numbers (sometimes subtract as well), all in accordance with a limited repertoire of binary word instructions. These chores are performed sequentially through a series of discrete steps. The secret to whether a problem is amenable to computer solution depends upon whether a plan of action (called an *algorithm*) can be written that will lead to a solution by a sequentially executed series of steps. Most practical instrumentation, control, or data processing chores can be so solved—a factor which accounts for the meteoric rise of the microprocessor. A field of endeavor that studies sequential solutions to practical (and some not so practical) problems is called *numerical methods*.

The MAD computer is merely a hypothetical construct used as a teaching aid. Let's examine a *real* microprocessor—the 6502.

2

6502 Architecture

The 6502 is one of the two most popular microprocessor chips on the market. Originated by MOS Technology, Inc., maker of the KIM-1 microcomputer, the 6502 is now available from more than 15 secondary sources. Among these secondary sources are Synertek and Rockwell International, who make the SYM-1 and AIM-65 microcomputers, respectively. The 6502 is widely used in applications which range from small Original Equipment Manufacturer (OEM) single-board computers and process controllers to elaborate data processing systems.

The 6502 is actually only one member (albeit the most popular member) of a family of microprocessor chips. Other members of the 65xx family include 6500/1, 6503, 6504, 6505, 6506, 6507, 6512, 6513, 6514, and 6515 devices. All members of the 65xx family (except 6502 and 6512) are housed in the 29-pin DIP IC package. The 6502 and 6512 come in the 40-pin DIP IC package. The 6502 and 6512 are very similar to each other, except that 6512 has a data bus enable (DBE) terminal which the 6502 lacks. The 6500/1 is a single-chip computer that includes, in addition to the CPM circuitry, internal ROM, read/write memory, two timers, and four 8-bit input/output ports. The 6500/1 recognizes several timer and I/O instructions in addition to the regular 6502 instruction set.

The two basic philosophies behind third-generation microprocessor architecture are: (1) register-oriented, and (2) memory-oriented. The popular Zilog, Inc. Z80 (which grew out of Intel's 8080a) is an example of a register-oriented microprocessor. The companion volume to this book, *Z80 User's Manual,* is available from Reston Publishing

Company. The 6502 grew out of the philosophy used to develop the 6800, and is an example of a memory-oriented machine.

The differences between the two philosophies are best seen in the structure of the I/O functions and the registers. The Z80 has numerous internal registers, while on the 6502, register functions are performed in external memory. Also, there are no Z80-like I/O instructions for 6502. All I/O ports are treated as memory locations. Such a system is often termed *memory-mapped I/O.*

The specific I/O instructions and internal registers of the Z-80-type chip are advantageous in some applications, but for the most part confer only little advantage over 6502-style systems. In fact, since 6502 can perform certain logical and arithmetic operations directly on memory (without the need for intervening data transfers), some types of program will execute considerably faster on 6502 than on Z80. Both types of chip architecture have their optimum applications, as witnessed by the huge success of *both* Z80 and 6502 devices.

Figure 2-1 shows the block diagram for the 6502. Like most microprocessors of the era, the 6502 uses an 8-bit bidirectional data bus (DB0-BD7) and a 16-bit address bus (A0-A15); the address bus is unidirectional (output). Since there are 16 bits to the address bus, the

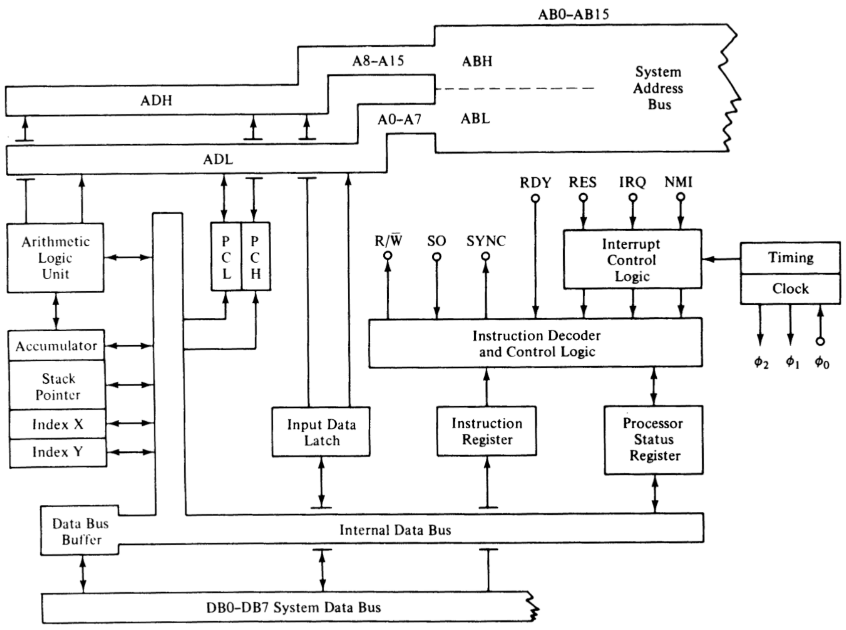


Figure 2-1. Block diagram for the 6502 microprocessor

6502 can uniquely address 2^{16}(65,536) different memory locations. Such a computer is a 65K machine.

6502 INTERNAL STRUCTURE

The 6502 is a complete central processing unit (CPU), which contains the following sections and registers: Arithmetic logic unit (ALU), accumulator (A-register), instruction register, instruction decoder/control logic section, interrupt control logic, processor status register, timing section, input data latch, stack pointer, index X register, index Y register, program counter (PCH and PCL), and data bus buffer. These are described here:

Arithmetic Logic Unit (ALU). The ALU is the internal logic that performs all arithmetic (ADC, SBC) and logical (AND, ORA, EOR) operations. The programmer does not have direct access to the ALU, except that the ALU is automatically implied by the instructions which affect the ALU.

The ALU is the heart of any CPU, and is one primary factor that distinguishes a computer or microprocessor from all other digital electronic circuits. This circuit performs the data manipulation including addition, subtraction, comparison, logical-AND, logical-OR, logical-XOR, left-shift, left-rotate, right-shift, right-rotate, bit set or reset, increment, decrement, and bit-test.

Accumulator. The accumulator, also called the A-register in some texts, is the main internal storage register in the 6502. Its function is to temporarily store data being operated on by the ALU. In the 6502, the accumulator is an 8-bit register that corresponds with the data bus on a bit-for-bit basis (i.e., bit 0 of the accumulator will travel to/from the CPU over DB0 of the data bus, bit 1 over DB1, etc.). Unless otherwise specified, all instructions executed by the 6502 use the accumulator. The addition-with-carry (ADC) instruction, for example, performs binary addition between an 8-bit data word fetched from memory and the contents of the accumulator.

Instruction Register (IR). The instruction register is the internal 6502 register where the instruction op-code is temporarily stored after it is fetched from memory.

Instruction Decoder/Control Logic. This section contains the logic circuits that will examine the contents of the instruction register, determine what operation is intended, and then permit the CPU to execute that instruction.

Interrupt Control Logic (ICL). An interrupt is a means by which an external device can gain control of the program. There are two active-LOW interrupt input lines on the 6502: NMI and IRQ. The NMI input is a nonmaskable interrupt. When NMI is brought LOW, the processor will switch control to a predetermined subroutine after the current instruction is executed. The IRQ is a maskable interrupt request input. Whether or not CPU recognizes the request is determined by the state (1 or 0) of an interrupt masking bit in the program status register. The programmer can cause IRQ to be enabled by executing SEI and CLI bits. The subject of interrupts will be discussed in greater detail in Chapters 12 and 13. For now, we will simply state that the logic for handling the interrupt function is the business of the ICL section. The system reset (RES) line also is part of the ICL. The system reset is activated manually by the user, or automatically by a power-on reset circuit. The RES line on many computers is nothing more than a hardware "Jump to 0000" instruction. On the 6502, however, RES is a *vectored jump*, meaning that it will jump to a location specified by the contents of memory locations FFFCH and FFFDH.

Processor Status Register (PSR). The PSR is an 8-bit/internal register that is used to indicate that status of certain processor functions. Each bit of the PSR is a "flag" and is independent of the other bits of the PSR. The flags tell the world the CPU status by being either set (1) or reset (0). The state of each flag is determined either by program control or by the result of the last operation. For example, the interrupt mask flag (I-flag) can be set or reset directly by SEI or SLI instructions, respectively. However, the Zero Flag (Z) is set or reset according to the results of operations on accumulator data. Arithmetic and logical instructions, for example, will leave Z = 0 if the result stored in the accumulator is non-zero, and Z = 1 if that result is zero. The six flags of the PSR are:

N	Negative result (bit 7 = 1)
Z	Zero result (all bits = 0)
C	Carry Flag (arithmetic produced a result, carry)
I	Interrupt mask flag
D	Decimal mode flag
V	Overflow flag

Timing Section. Computers operate synchronously with one or more system clocks. The 6502 uses three clock signals: Φ_1, Φ_2, and Φ_0. The Φ_1 and Φ_2 clock signals are internally generated by the timing section, and are available as outputs (see Chapter 3). The Φ_0 clock is the master system clock, and is generated externally to the 6502.

Stack Pointer (SP). The SP register contains the low order byte within Page One (0100H to 01FFH) where the stack is located. The *push* (i.e., PHA and PHP) and *pull* (i.e., PLA and PLP) instructions operate the stack. The higher order byte of the stack start address is always 01H, with the low order byte (00H to FFH) being supplied by the SP.

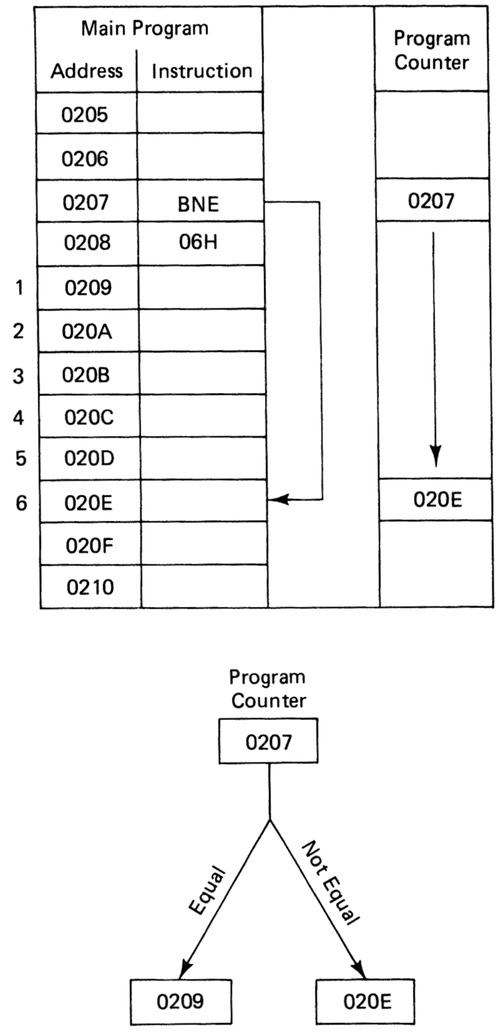
Index Registers X and Y. The X and Y index registers are 8-bit internal registers used in the indirect indexed addressing. In that form of addressing, the contents of either X or Y registers are added to a 2-byte address fetched from memory as a part of the instruction. The X and Y registers can also be operated on by certain instructions, such as load, store, increment, decrement, and exchange data.

Program Counter (PCL and PCH). The program counter is a pair of 8-bit registers which contain the address where the next instruction to be executed is stored in memory. When taken together, PCL and PCH form a 16-bit address. When the reset line on the 6502 is brought LOW, either by the power-on reset circuit or by a manual reset button, the program counter is loaded with the address bytes stored at locations FFFCH and FFFDH. In other microprocessors, the reset causes a jump to location 0000H.

The program counter is altered in several ways. Every time an instruction is executed, the program counter is incremented by the number of bytes required for that instruction: a 1-byte instruction increments PC by 1, a 2-byte instruction by 2, etc. For example, in Figure 2-2, the main program encounters an *add with carry* (ADC) instruction at location 0201H. This particular form of ADC uses a form of addressing in which the operand is stored at a location denoted by the 2 bytes following the ADC. The op-code is stored at location 0204H. Thus, the program counter increments directly from 0201H to 0204H as ADC is being executed.

Address	Instruction	Program Counter
0201	ADC	0201
0202	(Byte #1)	│
0203	(Byte #2)	↓
0204	(Next Instruction)	0204
0205		

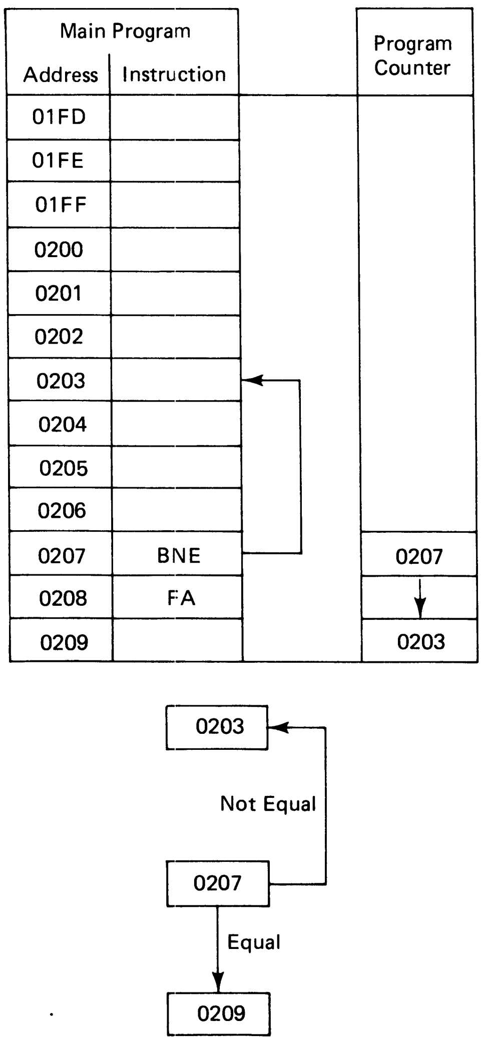
Figure 2-2. Operation of the program counter (an example)



(A)

Figure 2-3. Operation of the program counter during the BNE (branch on not-equal to zero) instruction A) forward branch.

Another way to alter the contents of the program counter is to execute a *branch* instruction such as BNE, BEQ, BCC, and BCS. These instructions use relative addressing. This term means that the program counter will be modified by an amount denoted by the second byte of the instruction. Forward branches are determined by using a positive hexadecimal number, while backward branches are denoted by a two's-



(B)

Figure 2-3 (continued). B) backward branch

complement equivalent negative hexadecimal number. For example, consider the *branch on result not equal to zero* (BNE) instruction shown in Figures 2-3A and 2-3B.

The BNE instruction examines the Zero Flag (Z) in the Processor Status Register for Z = 0, which indicates that the result of a previous operation was *not* equal to zero. BNE will fall through to the next instruction in sequence (e.g., 0209H in Figure 2-3A) if the result was zero (Z = 1). If the result was non-zero (Z = 0), then BNE forces a jump forward or backward a number of steps denoted by the second byte of the instruction. It does this neat trick by altering the program counter contents. Two situations are given in the figures; a forward branch is shown in Figure 2-3A, while a backward branch is shown in Figure 2-3B. Let's consider the forward branch first.

Figure 2-3 shows a forward branch BNE operation from location 0207H. The op-code for BNE is stored at 0207H and the operand 06H is a positive hexadecimal number, so the program will branch six steps *forward* when the branch condition (i.e., Z = 0 for BNE) is satisfied. Consider first the situation where the condition is not satisfied (Z = 1). When BNE is encountered, it reads Z to determine status (1 or 0). If Z = 1, then the condition is not satisfied, so the program "falls through" to the next instruction. Since BNE is a 2-byte instruction, the next location is 0207H +2, or 0209H. When the condition is not met, therefore, the program counter is incremented from 0207H to 0209H.

The alternate situation in Figure 2-3A is when the condition is satisfied (Z = 0). Since the second byte is 06H, the instruction BNE will cause a branch forward by six steps; the program counter is altered by +6 to 020EH. Notice that the six steps are counted from the *next* step following the BNE and its operand, i.e., 0209H is the base for the count, not 0207H.

The backward branch situation is shown in Figure 2-3B. The situation for condition not satisfied is exactly the same as the other case. The program counter will be advanced from 0207H to 0209H. For example, for a backward branch of six steps we would use the two's complement of −6, which is FAH, in the second byte. Counting from the address of the next instruction (0209H), six steps would bring us to 0209H; the program counter is altered to 0203H.

One final way to alter the program counter is to execute either a jump instruction or an interrupt. In both cases, the operation transfers control to some other memory location by altering the PC contents.

The 6502 program counter is divided into two 8-bit registers called PCL and PCH. The PCL register outputs the low byte of the 16-bit address, while PCH outputs the high byte of the address. PCL and PCH forms the 16-bit address.

MEMORY ALLOCATION RESTRAINTS

Memory space in microcomputers is usually divided into "pages" of 256 bytes each. Page zero is 0000H to 00FFH, page one from 0100H to 01FFH, page two from 0200H to 02FFH, etc. On the 6502, we are constrained from using locations in page zero, page one, and page FFH.

Page Zero. Memory locations from 0000H to 00FFH are in page zero, and are used in two different addressing modes: *zero page* and *indirect*. In zero page addressing, the CPU assumes that the high order byte of the address is 00H, while the low order byte is the second byte of the instruction. In indirect addressing, the second byte of the instruction points to a location in page zero where the low order byte of the intended address is stored; the high order byte will be stored at the next higher memory location. Since there are 256 locations in page zero, we can store up to 128 pairs of address bytes.

Page One. The "stack" is a section of memory used by the processor for such chores as the temporary storage of program counter contents when the processor goes to a subroutine. In the 6502, the stack is in page one (from 0100H to 01FFH). Usage of either page zero or page one addresses should be done cautiously because of these pre-emptory uses.

Page FFH. The six highest bytes in page FFH are predesignated for certain vectors, arranged in three pairs. These vectors are the addresses where the computer goes on reset and on both types of interrupt. These locations are pre-allocated as shown in Figure 2-4.

Memory Location	Use	Comment
FFFFH FFFEH	IRQ	Interrupt request line (IRQ) low causes processor to jump to the memory location specified in these two bytes. The high order byte of the 16-bit address is stored at FFFFH, while the low order byte is at FFFEH.
FFFDH FFFCH	RESET	Reset low causes jump to address specified by the contents of these locations: FFDH contains the high order byte, FFCH contains the low order byte.
FFFBH FFFAH	NMI	Nonmaskable interrupt request. See IRQ above. High order byte is stored at FFFBH, low order byte at FFFAH.

Figure 2-4. Vector locations for IRQ, RESET, and NMI

3

6502 Pinouts

The 6502 microprocessor is housed in a 40-pin Dual Inline Package (DIP). This package is shown in Figure 3-1 with the pinout designations that apply to the 6502. Note that most microprocessor chips use NMOS technology, so appropriate anti-static handling procedures must be followed lest the IC be zapped into never-never land.

6502 Pinouts by Pin Number

1	V_{ss}	0 to +7 volts
2	RDY	Ready
3	Φ_1 (out)	Phase-1 clock output
4	$\overline{\text{IRQ}}$	Interrupt request
5	N.C.	(no connection)
6	$\overline{\text{NMI}}$	Nonmaskable interrupt
7	SYNC	Synchronization
8	V_{cc}	+5 volts
9	AB0	Address bus bit 0
10	AB1	Address bus bit 1
11	AB2	Address bus bit 2
12	AB3	Address bus bit 3
13	AB4	Address bus bit 4
14	AB5	Address bus bit 5
15	AB6	Address bus bit 6
16	AB7	Address bus bit 7
17	AB8	Address bus bit 8
18	AB9	Address bus bit 9
19	AB10	Address bus bit 10

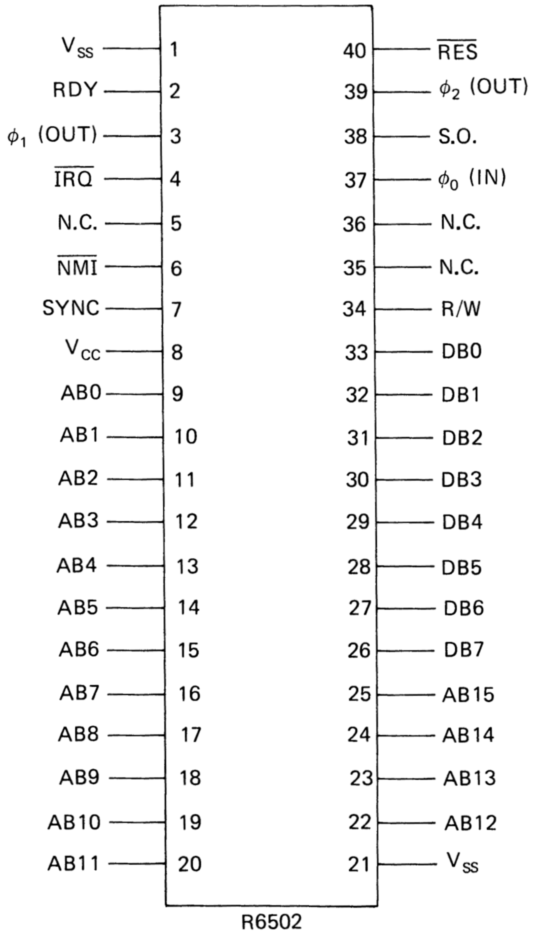


Figure 3-1. 6502 pinouts

6502 Pinouts by Pin Number (continued)

20	AB11	Address bus bit 11
21	V_{ss}	0 to +7 volts
22	AB12	Address bus bit 12
23	AB13	Address bus bit 13
24	AB14	Address bus bit 14
25	AB15	Address bus bit 15
26	DB7	Data bus bit 7
27	DB6	Data bus bit 6
28	DB5	Data bus bit 5
29	DB4	Data bus bit 4
30	DB3	Data bus bit 3

6502 Pinouts by Pin Number (continued)

31	DB2	Data bus bit 2
32	DB1	Data bus bit 1
33	DB0	Data bus bit 0
34	$R/\overline{W}$	$\text{Read}/\overline{\text{Write}}$
35	N.C.	(no connection)
36	N.C.	(no connection)
37	Φ_0 (IN)	Phase-0 clock input
38	S.O.	Set overflow flag
39	Φ_2 (OUT)	Phase-2 clock output
40	$\overline{\text{RES}}$	$\overline{\text{RESET}}$

6502 Pinout Descriptions

Designation	*Pin*	*Description*
DB0 – DB7	(below)	Eight-bit bidirectional data bus. LOW (logical-0) is V_{ss} to V_{ss} + 0.4 volt; input HIGH is V_{ss} + 2.4 volts to V_{cc}
DB0	33	data bus bit 0
DB1	32	data bus bit 1
DB2	31	data bus bit 2
DB3	30	data bus bit 3
DB4	29	data bus bit 4
DB5	28	data bus bit 5
DB6	27	data bus bit 6
DB7	26	data bus bit 7
AB0 – AB15	(below)	Sixteen-bit address bus capable of addressing up to 65,536 (64K) unique memory locations. These lines are all outputs, and produce the same HIGH and LOW voltage levels as the data bus lines will respond to; i.e., output-LOW (logical-0) is V_{ss} to V_{ss} +0.4 volt, while output-HIGH (logical-1) is V_{ss} +2.4 volt to V_{cc}
AB0	9	address bus bit 0
AB1	10	address bus bit 1
AB2	11	address bus bit 2
AB3	12	address bus bit 3
AB4	13	address bus bit 4
AB5	14	address bus bit 5

6502 Pinout Descriptions (continued)

Designation	*Pin*	*Description*
AB6	15	address bus bit 6
AB7	16	address bus bit 7
AB8	17	address bus bit 8
AB9	18	address bus bit 9
AB10	19	address bus bit 10
AB11	20	address bus bit 11
AB12	22	address bus bit 12
AB13	23	address bus bit 13
AB14	24	address bus bit 14
AB15	25	address bus bit 15
Φ_0	37	Phase-0 system clock input. Either an RC timing network (not recommended) or an external crystal clock oscillator will supply a 1 mHz signal (2 mHz in some versions) to this pin
Φ_1	3	Phase-1 clock output; generated internally from Φ_0 clock; complement of phase-2 clock
ϕ_2	39	Phase-2 clock output; generated internally from Φ_0 clock; complement of phase-1 clock
$R/\overline{W}$	34	Indicates the *direction* of the data on the data bus; when this line is HIGH, the CPU is processing a read (input) operation; when this line is LOW the CPU is processing a write (output) operation
$\overline{IRQ}$	4	Interrupt request, this active-LOW input is used to interrupt the program being executed so that a subroutine can be executed instead. This interrupt input is maskable, so it will cause a response only if the internal interrupt flag of the Processor Status Register is enabled
$\overline{NMI}$	6	Nonmaskable interrupt; similar to the interrupt request line ($\overline{IRQ}$), except that this active-LOW input is always active, and cannot be dis-

6502 Pinout Descriptions (continued)

Designation	***Pin***	***Description***
		abled by the programmer. Program will execute an interrupt subroutine instead of the main program as soon as the current instruction is finished execution
$\overline{\text{RES}}$	40	Reset; active-LOW reset input. Essentially a hardware jump instruction to a location in memory designated by a reset vector in page FFH
RDY	2	Ready; this signal is an input that will insert a wait state into the normal machine-cycle sequence. The RDY line is normally held HIGH, and must make a HIGH-to-LOW (negative-going) transition during the phase-1 = HIGH clock cycle in any operation other than a write
SO	38	Set overflow flag; this input will set (HIGH) the overflow flag if it makes a HIGH-to-LOW (negative-going) transition during the trailing edge of the phase-1 clock cycle
SYNC	7	Active-HIGH output that is used to indicate the instruction-fetch machine cycle
N.C.	5,35,36	no connection
V_{ss}	1,21	0 to +7 volts DC; usually grounded (0-volts)
V_{cc}	8	0 to +7 volts; usually +5 volts DC (makes system TTL compatible)

4 Timing and Control Signals

If your interest in computers is only to program in BASIC or assembly language, then you have little need to understand the workings of the chip. If, however, your needs and interests are in interfacing, computer design, or design of microprocessor-based instruments, then a thorough knowledge of the chip is necessary. Of critical importance are the control signals and timing system for the chip. These matters intimately affect design and interface efforts. For the 6502 microprocessor chip we need to consider the following:

- Data Bus
- Address Bus
- $R/\overline{W}$
- Data Bus Enable
- Ready
- Interrupt Request
- Nonmaskable Interrupt
- Reset
- Synchronization
- Set Overflow

DATA AND ADDRESS BUSES

The two independent buses on the 6502 microprocessor are data and address. Each of these buses is a multi-bit parallel data path; the data bus is 8-bits long, while the address bus is 16-bits long. Each bus operates with TTL-compatible voltage levels, which are:

Logical - 0 (LOW): 0 to + 0.8 volts
Logical - 1 (HIGH): +2.4 to + 5 volt

Each bus is a *parallel* path, so we find one 6502 terminal for each bit (see Figure 4-1). For both data and address buses, the 6502 will drive capacitance of at least 130 pF and one standard TTL input (i.e., it has a "fan-out" of 1 into 130 pF); one "TTL load" equates to a drive current of 1.8 mA at TTL voltage levels as given here, and is the specification for the load imposed by the input circuit of a TTL device. Thus, a fanout of 1 means the 6502 bus pins can each drive only one TTL device. To overcome this limitation, which 6502 shares with all other microprocessor chips, we must use high power bus driver chips between the 6502 and its two buses. These chips have a fan-in of 1, and fan-outs of 30, 100, or even 200. Most bus driver chips are arrays of noninverting TTL buffers.

The data bus consists of 8 parallel bits labelled DB0 through DB7. The data applied to DB0-DB7 must be stable (i.e., valid and unchanging) for the last 100 nanoseconds (100 nS) of the phase-two (2) clock pulse. The data bus is said to be *bidirectional* because data flows both into and out of the 6502 via this route.

The address bus consists of 16 parallel data tracks which carry the address of the location in memory where the data or instruction

Data Bus		Address Bus	
Bit	Pin	Bit	Pin
DBØ	33	AØ	9
DB1	32	A1	10
DB2	31	A2	11
DB3	30	A3	12
DB4	29	A4	13
DB5	28	A5	14
DB6	27	A6	15
DB7	26	A7	16
		A8	17
		A9	18
		A10	19
		A11	20
		A12	22
		A13	23
		A14	24
		A15	25

Figure 4-1. Address and data bus pinouts

is located. During phase-1 of the clock cycle, the contents of the program counter are output to address bus bits A∅ through A15. The data on the address bus are valid from 300 nS after the beginning of phase-1, and remain valid until the beginning of the next phase-1 cycle. The address bus is said to be *unidirectional* because data only flows in one direction, i.e., *from* the 6502 *to* memory. Since there are 16 bits on the address bus, the 6502 can uniquely address 2^{16}, i.e., 65,536 (64K) different memory locations.

R/$\overline{W}$ LINE

The read/write (R/$\overline{W}$) line tells memory and all who are interested whether a *read* or *write* operation is taking place. The line will be HIGH for a read, and LOW for a write. Like the bus lines, R/$\overline{W}$ line can drive one TTL load (i.e., 1.8 mA into 130 pF of capacitance).

The R/$\overline{W}$ line remains HIGH for all processor operations except a write. The operation of this line is coincident with the address bus, so all transitions on R/$\overline{W}$ line occur during the phase-1 clock pulse.

The R/$\overline{W}$ line is used in controlling the operation of memory I/O devices and other devices. This timing protocol will be discussed later in this chapter.

DATA BUS ENABLE (DBE)

The DBE line is not used on the 6502, but is used on the companion 6512 device. The DBE is found on pin 36 of 6512, which is N.C. (no connection) on 6502. This line is used to lengthen the phase-2 clock long enough for an external device to input data to the 6502. Most peripheral devices operate at slower speeds than the 6502, so will not be compatible unless a DBE signal, or software equivalent, is provided.

READY ($\overline{RDY}$)

The $\overline{RDY}$ line on the 6502 is similar in function to the WAIT line on the Z-80 chip. The function of the $\overline{RDY}$ line is to delay execution of a *read* operation long enough to permit slower devices to catch up. Certain types of memory—EPROMs, for example—have long access times. An older EPROM (1702A) has an access time of approximately 1 mS. This specification means that stored data will not be available at the EPROM output until 1000 microseconds after a stable address appears on the address bus and the chip select is activated. Since the 6502 operates at 1 mHz (on some versions, 2 mHz), the memory has to respond much faster than 1mS. The $\overline{\text{RDY}}$ line will cause the 6502 to delay, i.e., wait, to allow the slow memory device or peripheral to catch up.

Transitions on the $\overline{\text{RDY}}$ line should take place during phase-1, so they can be recognized during phase-2. $\overline{\text{RDY}}$ only affects read cycles. If the line is active (i.e., it sees a HIGH-to-LOW transition) during a write cycle, the 6502 will continue to function but will stop executing during the next read cycle.

INTERRUPT REQUESTS ($\overline{\text{IRQ}}$ AND $\overline{\text{NMI}}$)

The interrupt lines cause the program to cease executing the current program and switch instead to executing a secondary program. When either interrupt line goes LOW, the 6502 will:

1. Finish executing the current instruction.
2. Increment the Program Counter to the next location that would normally be used.
3. Push the address in the PC out to the internal stack so that it may be saved.
4. Jump to the location of the interrupt subroutine pointed to by vectors stored in Page-FF.
5. If the last instruction in the subroutine is RTI (return from interrupt), then the 6502 will retrieve the address stored on the external stack (in page-1) and return to the main program where it would have gone if no interrupt had occurred.

Interrupts are used for a variety of purposes including serving very slow peripherals, responding to alarms, or servicing devices or events which occur but rarely.

The two active-LOW interrupt lines on the 6502 are $\overline{\text{NMI}}$ and $\overline{\text{IRQ}}$. The $\overline{\text{NMI}}$ is a nonmaskable interrupt. When this line is brought LOW, the interrupt will occur regardless of anything the program has done. The other line $\overline{\text{IRQ}}$ (interrupt request) is maskable by the program. Before responding to $\overline{\text{IRQ}}$, the 6502 interrogates the interrupt disable (I) flag in the Processor Status Register. If the I-flag is set (HIGH), then the 6502 will not respond to $\overline{\text{IRQ}}$. The 6502 sets the I-flag whenever a reset is activated or when an SEI instruction is executed. The CLI instruction will reset the flag, and thereby enable the 6502 to respond to interrupts.

RESET ($\overline{\text{RES}}$)

The reset line forces the 6502 to initialize the PC at a location specified by reset vectors stored at locations FFFCH (low-order address byte) and FFFDH (high-order address byte). On most 6502-based microcomputers, there are two ways to activate (bring LOW momentarily) the $\overline{\text{RES}}$ line: manually (by pushbutton switch) and during the power-on period when power is first applied to the 6502.

The $\overline{\text{RES}}$ line is essentially a hardware *JUMP-Indirect* instruction whose argument is FFFCH and FFFDH.

SYNCHRONIZATION (SYNC)

The SYNC is an active-HIGH output signal that goes HIGH during phase-1 cycles in which an *op-code fetch* operation is taking place. The purpose of SYNC, therefore, is to identify op-code fetch cycles.

SET OVERFLOW ($\overline{\text{SO}}$)

The $\overline{\text{SO}}$ terminal will cause the overflow flag (V) of the Processor Status Register to be set (made HIGH). This active-LOW line looks for a HIGH-to-LOW transition, and is TTL compatible. The $\overline{\text{SO}}$ line is intended to work with special I/O interface chips, so will not normally be used elsewhere.

6502 CLOCK TIMING

The three clock pins on the 6502 are: Φ_0 (pin 37), Φ_1 (pin 3), and Φ_2 (pin 39). Programmable digital computers operate in a synchronous mode in which the master system clock keeps operations in proper step.

The phase-0 (i.e. Φ_0) clock is an input on the 6502, and receives the signal from an external clock oscillator circuit. The phase-1 and phase-2 clocks are derived from phase-0, and are complementary to each other. Figure 4-2 shows the relationship between phase-1 and phase-2. Since these signals are complementary, one will be HIGH (logical-1) when the other is LOW (logical-0), and vice versa. The clock outputs are TTL compatible, so are between 0 and 0.9 VDC when LOW, and between +2.4 and +5 VDC when HIGH. The total duration of these pulses—(Phase-1) + (Phase-2)—is the *cycle time* (T_{cycle}) of the 6502. If the normal clock speed of 1 mHz is used, then the cycle time is 1 μS (one microsecond) and each phase is 500 nanoseconds. In any

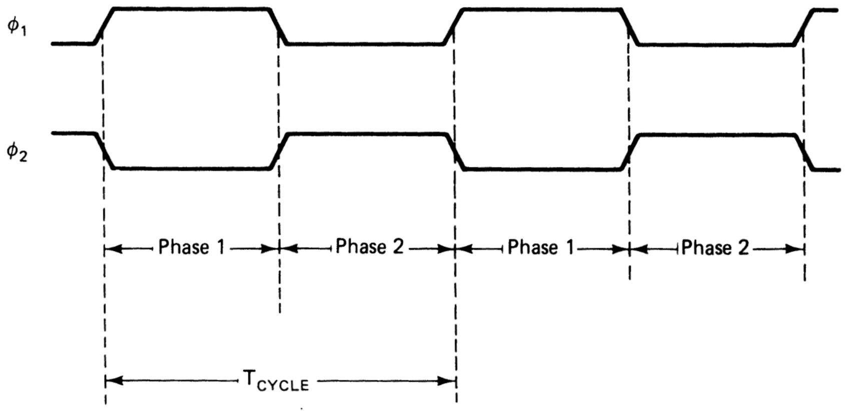


Figure 4-2. Two-phase system clock

given system, of course, the actual cycle time will be the reciprocal of the clock frequency—$T_{cycle} = 1/F_{clock}$. 6502 devices are available with clock frequencies up to 2 mHz, although the standard device operates at 1 mHz.

Figure 4-3 shows four different clock circuits used on 6502 microcomputers. Three of these clock oscillators are crystal controlled, while one is RC times. Crystals are piezoelectric devices which mimic the behavior of LC resonant tank circuits, and exhibit generally better frequency stability than RC networks. The RC version is sometimes preferred in low-cost applications, even though the unit cost of crystals is now low enough to make such considerations suspect except in the cheapest mass market products.

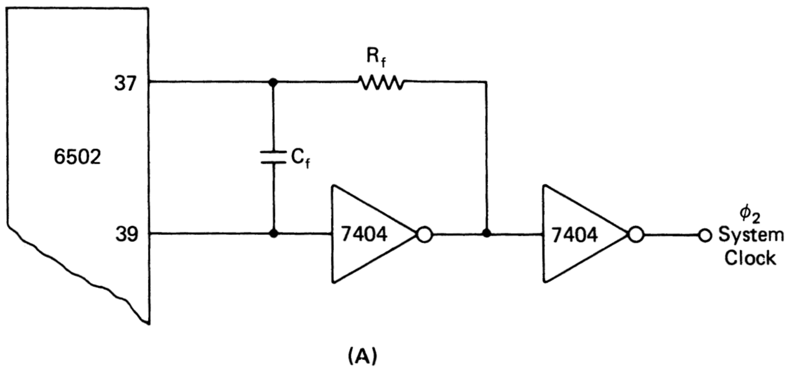
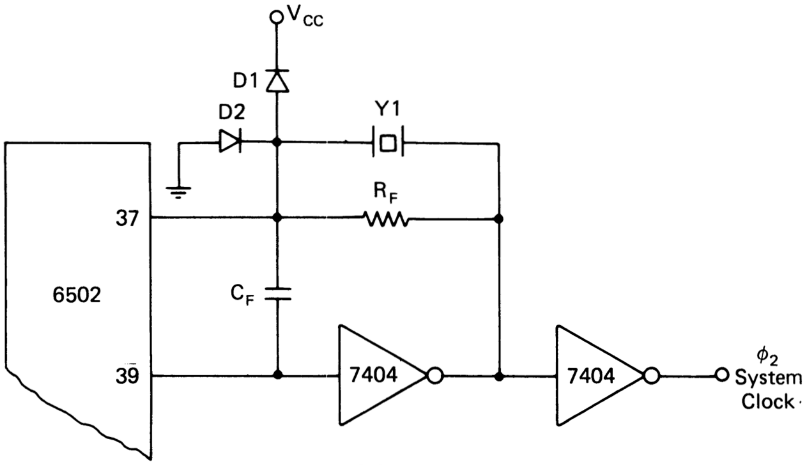
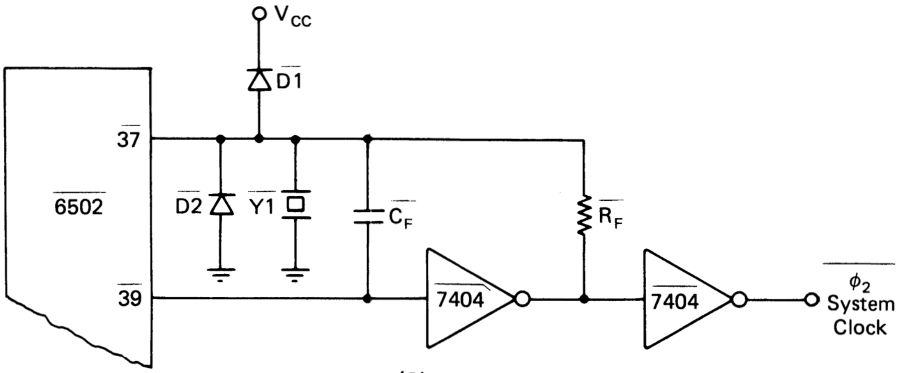


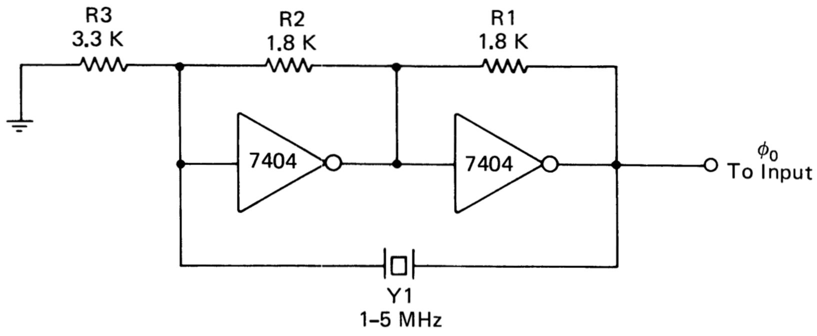
Figure 4-3. Typical main clock circuits A) RC operated.



(B)



(C)



(D)

Figure 4-3 (continued). B) crystal controlled, C) alternate crystal controlled, D) external crystal controlled. Y1: CTS-Night M-P Series (or equivalent)

The active element in each of the four circuits is a TTL inverter, in these cases a 7404 device. The 7404 is a hex inverter (six inverters in one package), of which two are used. Other TTL devices can also be used as inverters. If one or two 7400 or 7402 devices are available in the design, then they can be wired as inverters. The 7400 is a quad two-input NAND gate, while the 7402 is a quad two-input NOR gate. On either device, if both inputs of any one section are shorted together that section becomes an inverter. Additionally, the following circuit configurations also provide inverter action:

1. On 7400, tie one input HIGH (to +5 VDC through 2.7 kohm) and the remaining input works as an inverter input.
2. On 7402, tie one input LOW (to ground) and the remaining input works as an inverter input.

Figures 4-3A and 4-3B are connected to the phase-0 and phase-2 pins of 6502. The resistors will be between 0 and 500 kohms, while the capacitors are 2 to 12 pF.

Figure 4-3D shows a circuit that is totally external to the 6502. This circuit is also based on TTL inverters and is crystal controlled. The crystal used for this circuit, and certain other similar clocks, is a CTS-Knight MP-series device operating at 1 mHz, unless a 2 mHz CPU is used! The exact frequency is not critical unless a lot of timing loops in programming are anticipated.

READ/WRITE CYCLE TIMING

Data are input or output from the 6502 over the data bus during *read* and *write* operations. External memory and I/O port devices must be addressed and turned on at the appropriate time to supply or receive data. This operation is controlled by the system clocks and the R/W line. In this section we will discuss the action of the control signals during both forms of operation.

The timing for a write cycle is shown in Figure 4-4. Keep in mind that data direction statements are always made from the CPU point of view. Thus, a *data write* operation is the transfer of data *from* the CPU *to* some external memory location or I/O port. Since the 6502 uses memory-mapped I/O, the same write timing scheme serves for both *memory write* operations and writes to output ports.

When the CPU executes a *write* operation, the R/$\overline{\text{W}}$ line drops LOW and the address of the intended destination is output to the address bus. This action occurs during phase-1, which begins at time t_0 in Figure 4-4. Neither the R/$\overline{\text{W}}$ line nor the address are stable

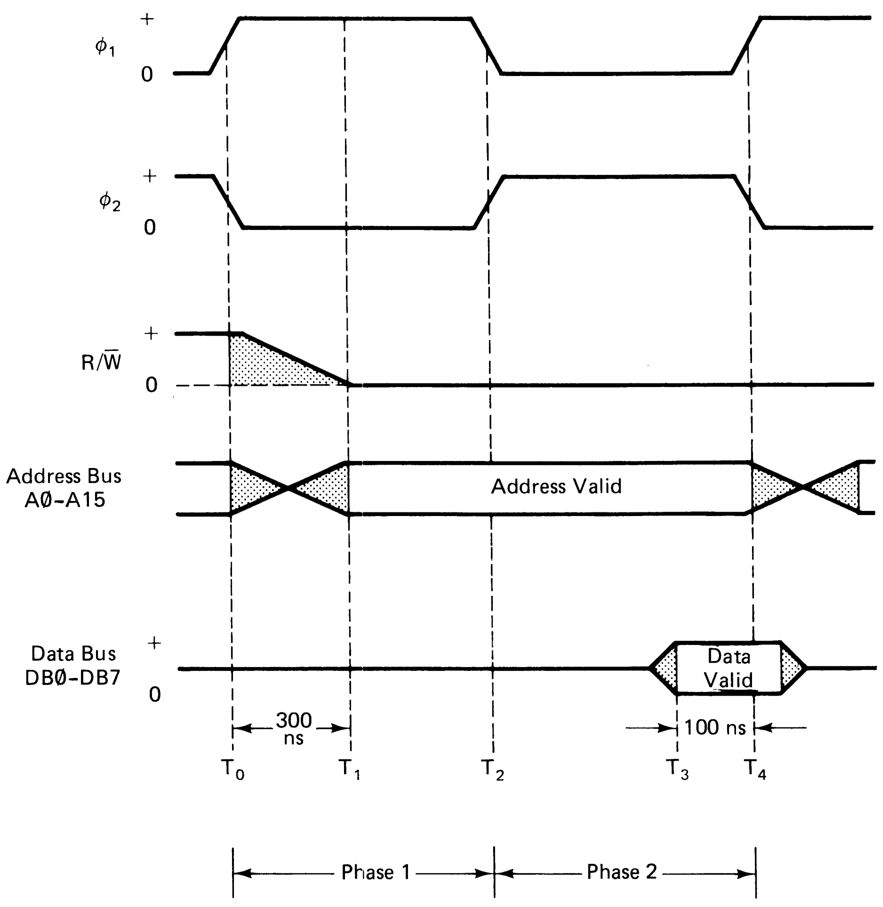


Figure 4-4. Write cycle timing diagram

immediately after the onset of phase-1, but rather require a time delay of about 300 nanoseconds (time t_1 in Figure 4-4). Following t_1, the address will remain valid, and the R/$\overline{W}$ line remains LOW during the remainder of phase-1 and all of phase-2. The actual data transfer takes place during the last 100 nanoseconds of phase 2. The entire cycle (sum of phase-1 and phase-2) requires 1 microsecond, or 1000 nanoseconds, when the clock operates at 1 mHz. In that case, the memory or I/O devices have approximately 575 nanoseconds between the initiation of a valid address and the onslaught of data from the 6502 to the data bus. This time period consists of the 1000 nS cycle time less address set-up time (300 nS), data valid time (100 nS), and transition times (about 25 nS). In a later chapter we will discuss address decoding and device select signal generation based on the waveforms of Figures 4-4 and 4-5.

The read cycle waveforms are shown in Figure 4-5. During a read cycle, data is transferred from some external memory location or I/O port to the 6502 CPU. The read cycle is exactly like the write cycle described above, except that the R/$\overline{W}$ line goes, or remains HIGH. The timing is otherwise approximately the same as for writes.

Logic circuits for selecting the memory or I/O device addressed will be discussed later.

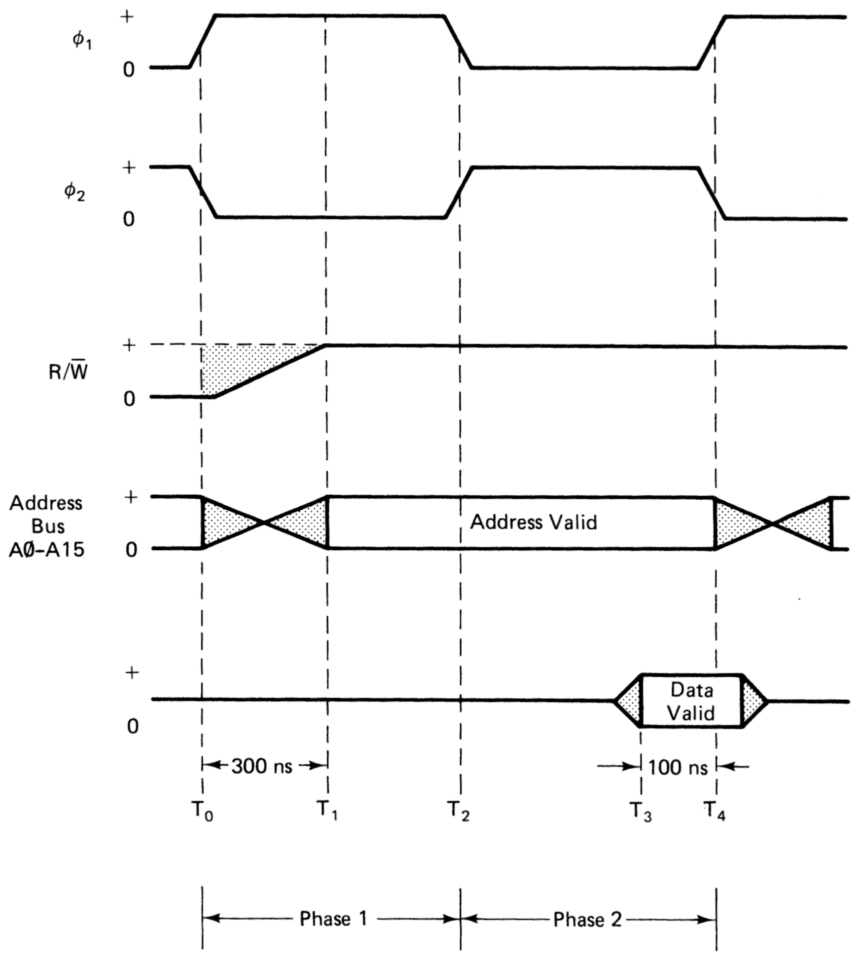


Figure 4-5. Read cycle timing diagram

5

6502 Addressing Modes

One way to judge the potential usefulness of a microprocessor is to examine the addressing modes, i.e., the number of different ways data can be addressed. Depending upon how you define *address modes,* we find the 6502 offering from 10 to 13 different modes. This fact makes 6502 either equal to Z80, or better by three modes.

Having a large number of addressing modes permits the programmer a certain degree of flexibility that is lacking on more limited processors. Figure 5-1 is a brief summary of 6502 addressing modes and the normal assembly language operand form. Following are summaries of the addressing modes for 6502.

ACCUMULATOR MODE ADDRESSING

The accumulator mode of addressing is an implied form that is unique to the *rotate* and *shift* instructions (ASL, LSR, ROR, and ROL). The *shift* instructions cause data in either the accumulator or a memory location to shift 1 bit right (LSR) or left (ASL). The *rotate* instructions are similar to the shifts, except that the rotated data is placed back into the accumulator and carry bit.

RELATIVE ADDRESSING MODE

Relative addressing mode is used for the *branch* instructions (i.e., BNE and BEQ). In relative addressing, the contents of the program counter are altered by a displacement factor, which can be either positive or

Accumulator	A
Relative	nn, nnnn
Immediate	nn
Absolute	nnnn
Zero Page	nn
Implied	—
Indirect Absolute	(nnnn)
Absolute Indexed, X	nnnn, X
Absolute Indexed, Y	nnnn, Y
Zero Page Indexed, X	nn, X
Zero Page Indexed, Y	nn, Y
Indexed Indirect	(nn, X)
Indirect Indexed	(nn), Y

("n" is a Hexadecimal Number Ø–A)

Figure 5-1. 6502 addressing modes

negative. The purpose of this mode is to allow shift of program control using only a 2-byte instruction. The first byte is the op-code, while the second byte is a signed two's complement number that represents the displacement integer *e*. Since this is a 2-byte instruction, and the branch cannot occur until the instruction is finished, the program counter will increment twice before the branch occurs. This accounts for the difference between the two jump ranges (+127 and −128).

Let's consider some examples. In both, the effective address is computed by adding the displacement integer *e*, the second byte, to the program counter at the *end* of instruction execution. Figure 5-2 shows an example of a forward branch operation. The instruction consists of 2 bytes at 0200H and 0201H; the op-code is at 0200H, while the displacement integer is at 0201H. The jump will occur at the end of instruction execution, if the condition for the branch is satisfied. If the condition is not satisfied, then the program counter contents will be 0202H. In this particular example, the displacement integer *e* is 06H, which designates a forward branch of +6. If the condition is satisfied, the program counter will jump to 0206H. This means that the next instruction to be executed will be that at 0206H. The maxi-

```
0200 (Byte-1) OP-CODE
0201 (Byte-2) 06H
0202 (PC if Condition not Satisfied)
0203
0204
0205
0206 (PC if Condition is Satisfied)
0207
0208
```

Figure 5-2. Forward branch operation

mum values allowed for the displacement integer are 7FH ($+127_{10}$) for forward branches, and 80H (-128_{10}) for backward branches.

The backward branch situation is shown in Figure 5-3. The 2-byte instruction is located at 0208H and 0209H; the op-code is at 0208H, while the displacement integer is at 0209H. The jump will occur when this instruction has completed execution. If the condition for the branch is not satisfied, then the program "falls through" to location 020AH; the program counter will then contain 020A rather than 0208H. But if the condition is satisfied, the program counter will contain the backward branch displacement integer, which in this case is the two's complement of -5_{10}, or FBH.

IMMEDIATE ADDRESSING MODE

The immediate addressing mode permits the use of a 2-byte instruction to operate on either the accumulator or an index register. Examples of instructions which have immediate mode addressing are ADC and LDA. The second of the 2 bytes is used as the operand, and is therefore the data which operates on the contents of the register or accumulator addressed; no additional data fetches from memory are needed. The mnemonic form used to write immediate mode instructions is ADC,

```
0200
0201
0202
0203
0204
0205 (PC if Condition is Met)
0206
0207
0208 (Byte-1) OP-CODE
0209 (Byte-2) FBH (−5)
020A (PC if Condition is not Satisfied)
```

Figure 5-3. Backward branch operation

n or ADC,#n (the latter is preferred in order to distinguish *immediate mode* from *relative* or *zero page mode* instructions).

We used ADC and LDA as examples here. Let's see how those instructions work with immediate mode addressing. The LDA instruction loads accumulator with data. In LDA, #n, the operand n is the next byte in sequence following the op-code:

Byte 1	op-code
Byte 2	(n)

If we load the hexadecimal number 80H into the accumulator, what would the instruction look like? Since A9H is the op-code for LDA when immediate addressing is used, we would see:

	Location	*Code*	
Byte 1	0500	A9	op-code
Byte 2	0501	80	(n = 80H)

assuming this program segment is stored at 0500H in memory.

This program segment will load the accumulator with the hexadecimal number 80H. Since this 2-byte instruction only requires two clock cycles, it will execute in only 2 μS.

The ADC instruction adds a data byte from memory to the contents of the accumulator, and generates a carry if indicated. The op-code for ADC when immediate mode addressing is used is 69H. Let's assume that the accumulator contains A7H when the following code is encountered:

	Location	*Code*	
Byte 1	0500	60	ADC, #n
Byte 2	0501	07	n=07H

This program segment means that the instruction fetched (69H) is the ADC,#n instruction, and that operand n is the next sequential memory location, 07H. After the execution of this 2-byte instruction, the contents of the accumulator will be:

$$\text{Acc} = (\text{Acc})+\text{n}$$
$$\text{Acc} = \text{A7H} + \text{07H}$$
$$\text{Acc} = \text{AEH}$$

Immediate mode addressing permits addition by a constant when ADC is used.

ABSOLUTE ADDRESSING MODE

The absolute addressing mode used to provide an operand from any location within the 64K is addressable by the 6502. The mnemonic form of the 3-byte absolute mode instruction is (using the ADC): "ADC, nnnn." As usual, the "n" represents a hexadecimal digit (a hexadecimal digit represents a 4-bit binary number). Since 4 bits is half a byte, some wags call it a "nybble." A 4-digit hexadecimal number—e.g., nnnn—represents a total of 16 bits needed to address 64K of memory.

The operand is fetched from a memory location determined by the 3-byte instruction. The first byte of the instruction is the op-code, which tells the computer what is to be done and which addressing mode is used. The second byte is the low-order byte of the address, while the third byte is the high-order byte of the address. For example, the address EF05H will be stored in the format:

Byte 1	(op-code)	
Byte 2	05H	Low-order address
Byte 3	EFH	High-order address

Consider the example in Figure 5-4. Here, we are instructing the 6502 to load the accumulator with the contents of memory location EF05H, which is the hex number 80H. The operation is:

1. The program encounters the instruction LDA, EF05H at 0600H. CPU goes to EF05H to retrieve number.

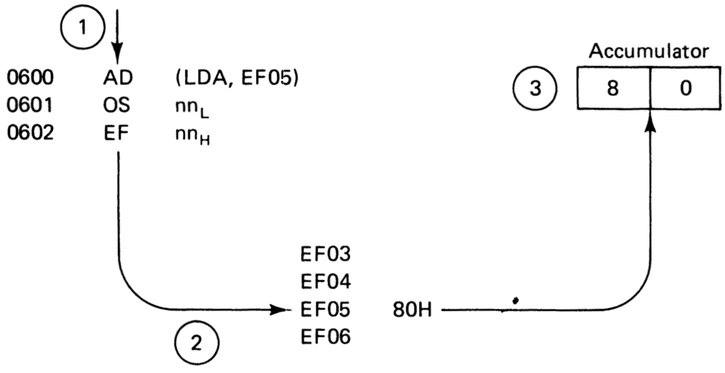


Figure 5-4. Absolute addressing mode example

2. Data from EF05H is fetched.
3. It is stuffed into the accumulator.

When the operation is finished, the hexadecimal number 80H will be in the accumulator.

IMPLIED ADDRESSING

In the implied addressing mode there is no external operand, and the address is implied by the instruction. For example, the *decrement x* (DEX) instruction causes the contents of the x-register to be decremented, i.e., reduced by 1; implied is the x-register. No additional addressing is needed to identify the data because it is the contents of X.

Other examples of instructions which use implied addressing are:

BRK	Force Break
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
DEX	Decrement X
DEY	Decrement Y
INX	Increment X
INY	Increment Y
NOP	No operation
PHA	PUSH Accumulator on Stack
PHP	PUSH PSR on Stack
PLA	PULL Accumulator from Stack
PLP	PULL PSR from Stack
RTI	Return from Interrupt
RTS	Return from Subroutine
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Bit
TAX	Transfer Accumulator to X
TAY	Transfer Accumulator to Y
TSX	Transfer SP to X

TXA	Transfer X to A
TXS	Transfer X to SP
TYA	Transfer Y to A

ZERO PAGE ADDRESSING MODE

The zero page addressing mode is an abbreviated-absolute mode that uses only 2 bytes to designate memory locations in page zero (0000H to 00FFH). In this mode, the high-order byte designating the address is always 00H. Thus, to address location 0052H using zero page addressing, we would use the form:

Byte 1	op-code	
Byte 2	(n)	00H-FFH

Note that we may still use absolute mode addressing in page zero. The advantage of zero page addressing is that, for the first 256 bytes of memory, we can use a more rapid 2-byte instruction. The main function is to reduce the program time, especially for frequently called data.

Let's use ADC for our example. The op-code for ADC in the zero page addressing mode is 65H. Figure 5-5 illustrates an example where

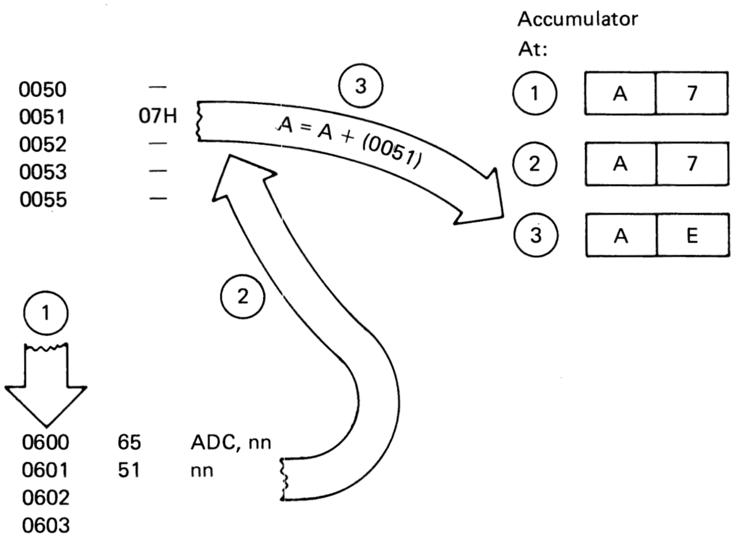


Figure 5-5. Zero page addressing mode example

the contents of the accumulator (A7H) are added to the contents of memory location 0051H. Our rotation for this operation is:

$$\text{Acc} = \text{Acc} + (0051\text{H})$$
$$\text{Acc} = \text{A7H} + 07\text{H}$$
$$\text{Acc} = \text{AEH}$$

(Note: The parenthesis around "0051H" indicate the *contents* of memory location 0051H). The operation is:

1. The program encounters an ADC, 51H zero page addressing instruction at location 0600H; the op-code (65H) is at 0600H and the zero page address is at 0601H. The contents of the accumulator are A7H.
2. The 6502 responds to the instruction by going to location 0051H and retrieving the data stored there (i.e., 07H).
3. The 6502 adds the contents of the accumulator (A7H) to the number fetched from memory location 0051H (i.e., 07H) and stores the result (AEH) in the accumulator.

Page zero should not routinely contain programming instructions in complex programs because the zero page addressing mode makes page zero an ideal place to store frequently called data, temporary data, short tables, and other data.

INDIRECT ABSOLUTE ADDRESSING MODE

This addressing mode is a subset of absolute mode, but is used only by the unconditional JUMP (JMP) instruction; JMP also uses absolute addressing. The indirect absolute mode is a 3-byte instruction with the mnemonic form:

JMP, (nnnn)

The operand (nnnn) is a 16-bit address at which the actual "jump-to" address is located. The low-order byte of the destination address is that actually specified by (nnnn), while the high-order byte of the destination is found at the next higher address (nnnn +1). Thus, if we want to store the destination address at memory location EF05H, we would write the JMP instruction as follows:

Byte 1	6C	Op-code for JMP (indirect)
Byte 2	05	(nn_L)
Byte 3	EF	(nn_H)

Let's consider an example (see Figure 5-6). Suppose we need to use indirect absolute addressing for a JMP instruction located at 0601H, and the destination address is located at EF05H. Let's see what happens:

1. At 0601H, the program counter is JMP (EF05). At the end of this instruction execution, the PC will contain "EF05."

2. In response to the change in PC contents, the computer will jump to EF05H where it finds the actual destination address, A008H.

3. The computer now jumps to A008H where an LDA #47H instruction is encountered. The PC now contains hexadecimal number 47H. When execution of this instruction is completed, the PC contents will be A008H +2, or A00AH.

Why would we want a program in which indirect absolute addressing is used? Isn't absolute addressing more reasonable? Under many circumstances that would be true. The exception is where the "jump-to" address changes under program control, perhaps in response

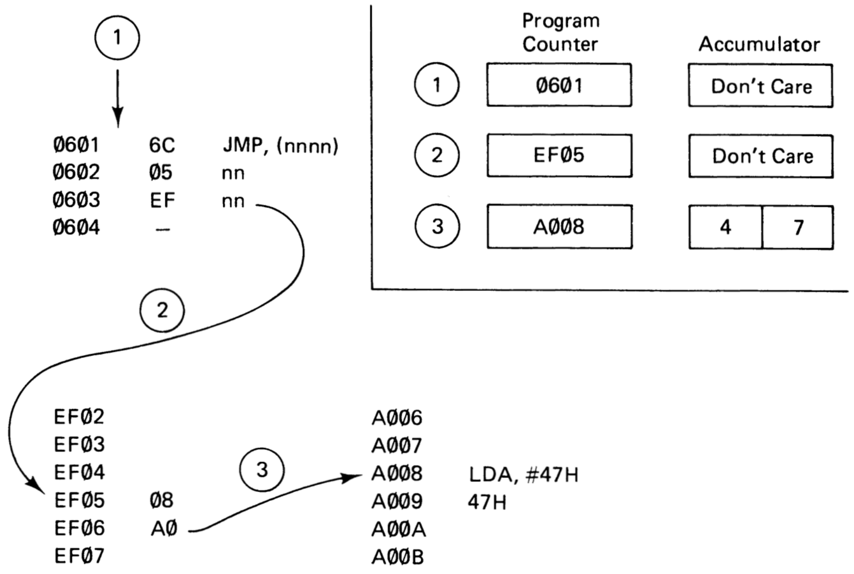


Figure 5-6. Indirect-absolute addressing mode example

to different conditions. Consider the hypothetical security alarm in Figure 5-7. Here we have a computer monitoring both fire and burglar alarm sensors. Obviously, the response to a fire on the premises requires a different response than to an intruder. Three different conditions can occur:

1. Fire alarm.
2. Burglar alarm.
3. No alarm.

The sensors are designed to input to the computer a hexadecimal number that serves the function of identification. This hex number code serves as the high-order byte for the subroutine that services that sensor. Note that either hardware initiated or software initiated schemes are used to obtain the code. The operation of this program is:

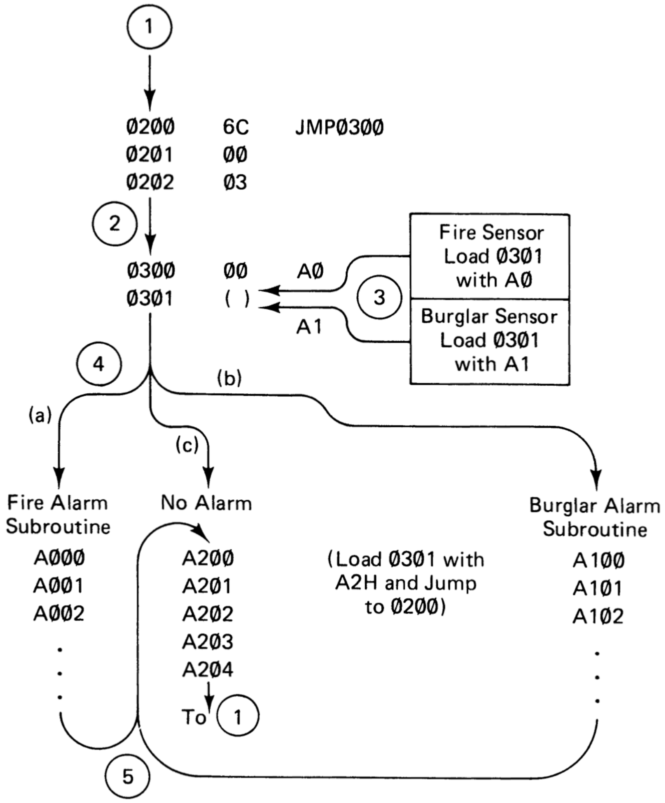


Figure 5-7. "Fire alarm" problem

1. The program encounters a JMP (indirect absolute) instruction at 0200H.
2. The operand for the JMP at 0200H is 0300H, so the program goes to 0300H to fetch the actual address of the destination. The low-order byte (at 0300H) is always 00H, while the high-order byte is determined by the sensor that is active.
3. The sensor has input appropriate code.
4. The program branches to either A000H, A100H, or A200H, depending upon course of action required.
5. If either the burglar or fire alarms were activated, the program will go to the no-alarm subroutine to reset the system after the alarm clears.
6. The sequence starts over.

In addition to these examples, the JMP (nnnn) mode can be used to designate peripherals. If we label the peripheral ports 1, 2, 3 . . . etc., then we can load the JMP operand bytes with the address of the subroutine that services that particular device.

ABSOLUTE INDEXED X AND Y ADDRESSING MODE

The absolute indexed addressing mode is used for such purposes as accessing data stored in an array or look-up table. The effective address of the instruction using this mode is the *sum* of the operand and the contents of either X or Y index registers. The form of such instructions is:

Byte 1	op-code	
Byte 2	(nn_L)	low-order address byte
Byte 3	(nn_H)	high-order address byte

The actual address of the data will be either:

$$nnnn + X$$

or,

$$nnnn + Y$$

If, for example, the X register contains 24H, and the instruction "LDA, 0400X" is encountered, the accumulator will be loaded with the data stored at 0400H + 24H, or 0424H. Another example (shown in Figure 5-8) is:

1. At 0200H the program encounters LDA EF00X.
2. The contents of the X register (08H) are added to EF00H to form the effective address EF08H.
3. Program goes to EF08H to fetch data (43H) to be stuffed into the accumulator.
4. Data is loaded into the accumulator (accumulator contents are now the data which had been at location EF08H).

There are several uses for the absolute indexed addressing mode, especially where tables or data arrays are concerned. A sample application is code conversion. Most modern computers use ASCII code to represent alphanumeric characters. ASCII is a 7-bit code (b0-b6), with the eighth bit (b7) always LOW. But suppose we want to interface an old Baudot-encoded teletypewriter (TTY) machine to an ASCII computer? The solution is a code conversion subroutine.

Figure 5-9 shows the flow of a code conversion program that takes an ASCII symbol from a keyboard and then converts it to a Baudot word that represents the same character (in this case "Q") before outputting it to a printer or TTY (Note: "Q" is represented by 51H in ASCII and 17H in Baudot). Since this scheme is an ASCII-to-Baudot routine, the argument of the "LDA, 0800X" instruction is the

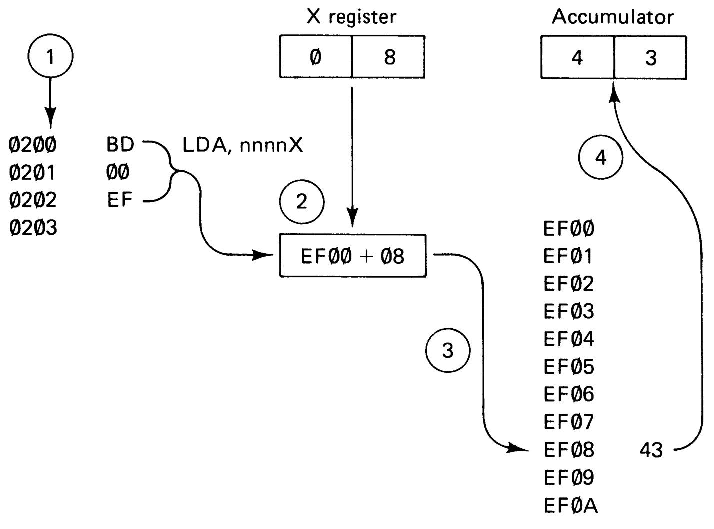


Figure 5-8. Absolute indexed (X or Y) addressing mode example

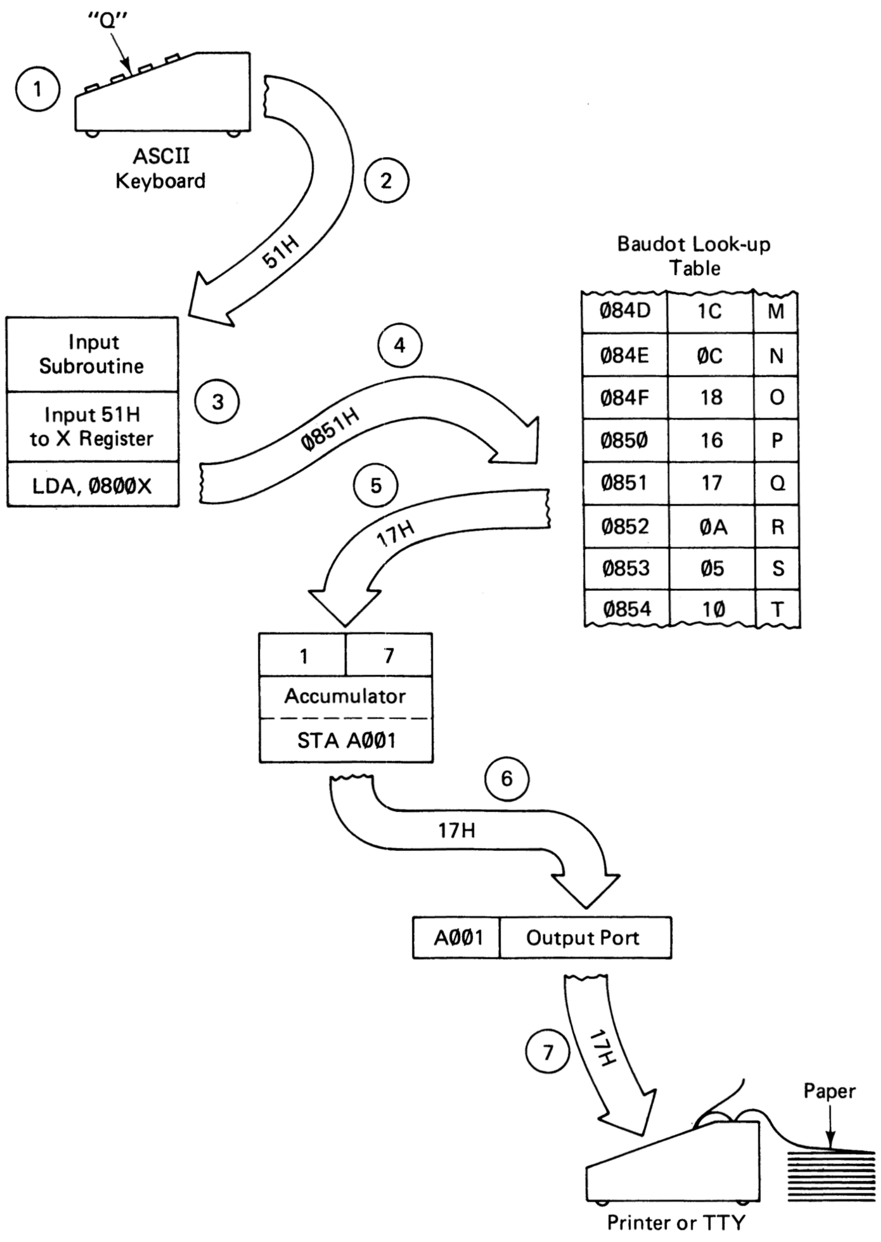


Figure 5-9. Code conversion scheme

ASCII for Q, i.e., 51H. The look-up table is stored in page 8 so that a character's Baudot representation will be located at 0800H + ASCII. Thus, since "Q" in ASCII is 51H, the Baudot code for "Q" (17H) will be located at 0800H + 51H, or 0851H. The operation of this program is:

1. Operator presses the "Q" on the keyboard . . .
2. Thereby sending 51H to the computer.
3. Since the 6502 uses memory mapped I/O, we can use the LDX instruction to directly input 51H from the port serving the keyboard to the x-register.
4. The computer sees an "LDA, 0800X" instruction so fetches data (17H) from 0800H + 51H (i.e., 0851H.)
5. Data from 0851H (17H) is loaded into the accumulator.
6. The 17H data loaded into the accumulator is stored at A001H, which is the memory location for the output port serving the printer/TTY machine.
7. The TTY machine sees 17H and responds by typing a "Q".

Code conversion is not the only application of a table look-up routine. We can also use this method to process data arrays. For example, we could input up to 256 bytes of data in say, page 7, i.e., 0700H to 07FFH. The X-register could be loaded with a number up to FFH, and then be decremented sequentially until the X-register contains 00H. After each datum from 0700H-07FFH is fetched, it is processed and another datum is fetched (see Figure 5-10).

The system shown in Figure 5-10 is an supersimplified "evoked potentials" computer. In evoked potentials studies, a stimulus (e.g., light flash) is applied to the subject repeatedly. By averaging the EEG (brainwave) signal in a time-coherent manner, we can eliminate the randomness and lull out only that portion of the signal which is due to the stimulus. The idea is to average or sum the data occurring at the same interval after the stimulus with each other. Thus, we must average all S+10 ms data together, all S+11 ms data together, S+12 ms, S+13 ms, etc., to 500–1000 ms. The system shown in Figure 5-10 is to average all sequential data at the same post-stimulus instant (for example, S+100 ms). The operation is:

1. Analog data is continuously acquired and converted (A/D) to a representative binary 8-bit word and is input to the computer.

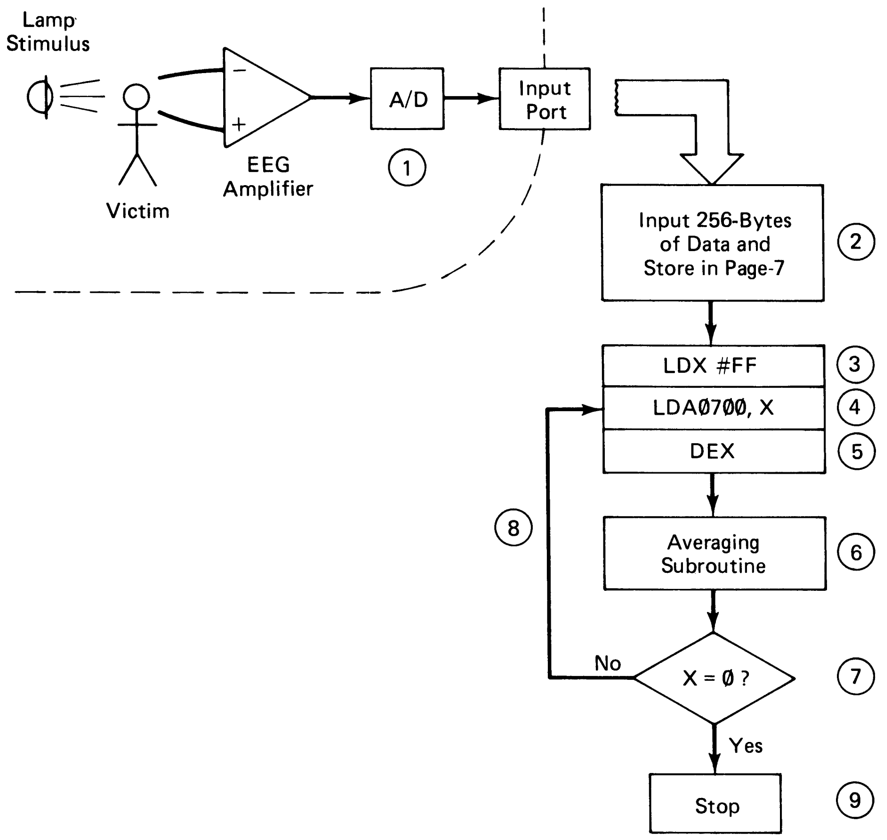


Figure 5-10. Hypothetical evoked potentials application example

2. An array of 256 samples of S+100 ms data are input and stored sequentially in 0700H to 07FFH.

3. The index register X is loaded with #FFH.

4. Data is fetched from 07FFH and stored in the accumulator.

5. The X-register is decremented by 1, becoming FFH−1H=FEH.

6. Data in the accumulator is used in an averaging subroutine.

7. The X-register is checked for 0 (BNE instruction used).

8. Since $X \neq 0$, the program branches back to pick up data from 07FFH−1H, or 07FEH. This looping continues until all data is processed, as indicated by $X = 0$ condition.

9. $X = 0$, so program is ended.

ZERO PAGE INDEXED (X AND Y) ADDRESSING MODE

Zero page indexed addressing is a subset of indexed addressing that uses a 2-byte instruction to designate locations in page 0 (i.e., 0000 to 00FFH). The effective address is calculated by adding the contents of either the X or Y index register to a base location specified by the second byte of the instruction. The form of the instruction is:

Byte 1	op-code	
Byte 2	nn	(page 0 base address)

The actual effective address will be either:

00nnH + X (e.g., LDA nn,X)

or,

00nnH + Y (e.g., LDA nn,Y)

depending upon which index register is specified by the op-code. For example, suppose we encounter LDA 50, X when the contents of the X-register are 0AH. The effective address is:

00nnH + X

0050H + 0AH = 005AH

Like its cousin, the zero page indexed addressing mode is particularly useful for lists, arrays, and tables.

INDIRECT INDEXED ADDRESSING MODE

This addressing mode combines the indirect with the indexed mode. In this mode, the effective address is calculated from the contents of a location in page zero that is pointed to by an indirect zero page instruction. The form is:

Byte 1	op-code	
Byte 2	nn	(page zero address)

The operand (nn) is a location in page zero where the low-order byte of the indirect address is stored; the high-order byte is stored at the next higher location. For example, LDA (40), X means that the low-order byte is at 0040H and the high-order byte is at 0041H.

Consider the example in Figure 5-11. The flow is as follows:

1. At 0200H the LDA (52), X" instruction is encountered.
2. The program jumps to 0052H where it finds the address EF05H.
3. The indirect address EF05H is combined with the contents of the X-register to . . .
4. Form the sum EF05H + 03H = EF08H.
5. The effective address is EF08H, so the processor goes to that location and fetches the contents (3AH).
6. Since this is an LDA instruction, the contents (3AH) of EF08H are stuffed into the accumulator.

The technique of indirect indexing is called post indexing.

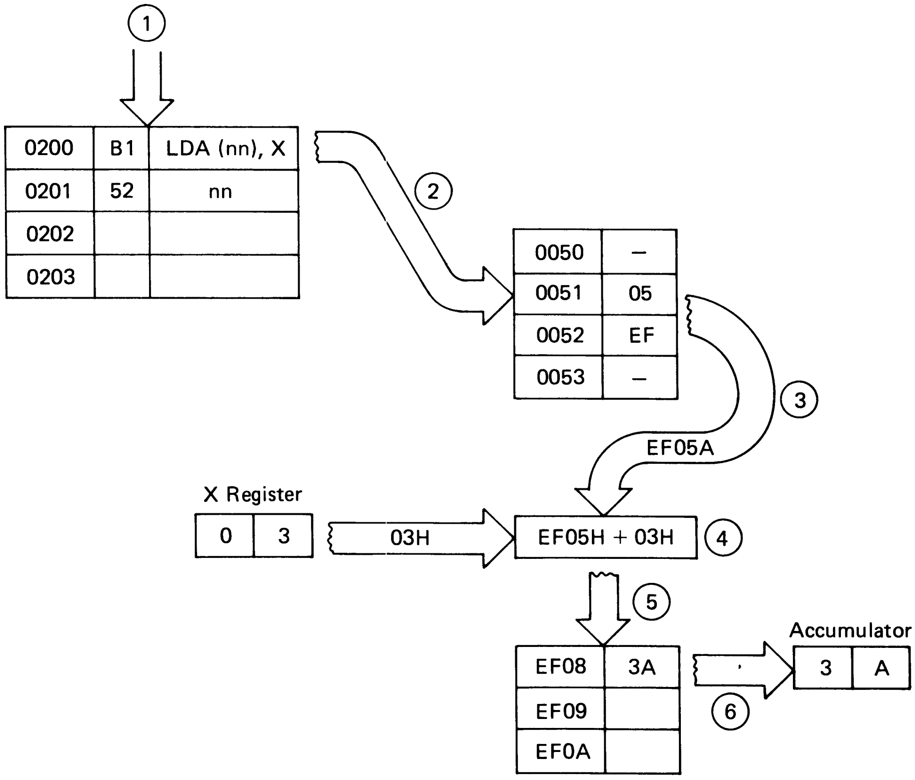


Figure 5-11. Indirect indexed addressing mode example

INDEXED INDIRECT ADDRESSING MODE

This method is related to indirect indexed addressing, except that the contents of the index register (X or Y) are added to the zero page address specified by the second byte of the instruction. The form is:

Byte 1	op-code	(e.g., LDA (nn,x)
Byte 2	nn	

The effective address is the contents of memory location nn+X or nn+Y. This is usually written:

$$(nn + X)$$

and

$$(nn + Y)$$

The parentheses mean "the contents of . . ." the argument inside ( ).

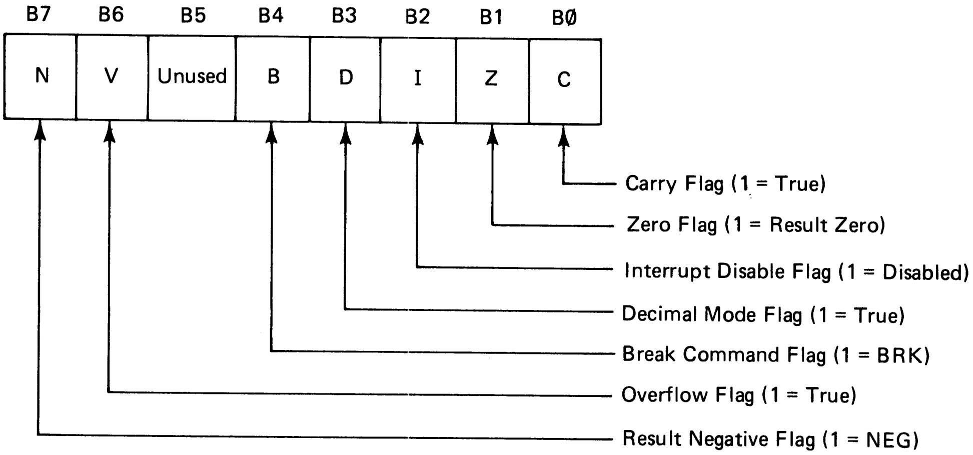
6

6502 Status Flags

The processor status register (PSR) is an 8-bit internal 6502 register which contains information concerning the results of previous operations. Figure 6-1 shows the details of the PSR, which are summarized here.

FLAGS

Negative (N) Flag. The N-flag is used by the 6502 to indicate that the result of executing an instruction is negative. The value of the



B7
B6
B5
B4
B3
B2
B1
BØ
N
V
Unused
B
D
I
Z
C
Carry Flag (1 = True)
Zero Flag (1 = Result Zero)
Interrupt Disable Flag (1 = Disabled)
Decimal Mode Flag (1 = True)
Break Command Flag (1 = BRK)
Overflow Flag (1 = True)
Result Negative Flag (1 = NEG)

Figure 6-1. 6502 Processor status register flags

N-flag is always equal to the value of the MSB (bit 7) in the accumulator. Thus, the N-flag can also be used in any operation which results in a change in bit 7. A common example is inputting the 7-bit ASCII data from a keyboard or other peripheral. In that case, bit 7 will be used as a strobe to let the computer know that new data are available. The N-flag can be used to record this fact. The following instructions affect the N-flag:

ADC	CPY	EOR	LDX	PLP	TAX
AND	CPX	INC	LDY	ROL	TAY
ASL	DEC	INX	LSR	ROR	TSX
BIT	DEX	INY	ORA	RTI	TXA
CMP	DEY	LDA	PLA	SBC	TYA

The N-flag cannot be directly affected by the user, but there are schemes which programmers can use to indirectly affect the N-flag. Since LDA, LDX, and LDY affect the N-flag, we can perform a dummy load operation whose only purpose is to set the N-flag. If, for example, the Y-register is not being used, then we can set N = 0 using "LDY, #00" or to N = 1 by using "LDY #80H." If all three registers are being used, then we can temporarily store the contents of the selected register somewhere in page zero of memory, to be retrieved after the dummy load is completed. We must be sure that the restoration does not alter the N-flag.

Overflow (V) Flag. The V-flag can be used in two ways by the 6502. First, it is used in signed binary arithmetic operations to indicate that the result could not be stored in the low-order 7 bits of the accumulator. Second, the V-flag is used with the BIT instruction. In that case, the V-flag is set to the value of B6 in the accumulator. The following instructions will affect the V-flag:

For signed binary: ADC, SBC

For other operations: BIT, CLV, PLP, and RTI

Break Command (B) Flag. The B-flag will indicate whether an interrupt was the result of a BRK instruction or the result of an interrupt signal from the outside world. Only the BRK instruction affects the B-flag. The B-flag will be HIGH (i.e., B = 1) if a BRK is executed, and LOW (i.e., B = 0) at all other times.

Decimal Mode (D) Flag. The D-flag indicates whether the CPU is operating as a straight binary adder or as a binary coded decimal (BCD) adder. If the D-flag is set (i.e., D = 1), the D-flag is reset, and the 6502

CPU is in binary mode. The instructions which affect the D-flag are: SEO, CLD, RTI, and PLP. The SEO (set decimal mode) causes the D-flag to be directly set to 1; the CLD causes the D-flag to be directly reset to 0.

Interrupt Disable (I) Flag. The I-flag is used to mask or permit the operation of the interrupt request (IRQ) line. If the I-flag is set (i.e., I = 1), then the 6502 will ignore interrupt requests on the $\overline{\text{IRQ}}$ line (Note: interrupt requests made on the $\overline{\text{NMI}}$ line are nonmaskable, so are not affected by the I-flag status). If, on the other hand, the I-flag is reset (I = 0) then the 6502 will honor interrupt requests. A low on $\overline{\text{IRQ}}$ will cause the 6502 to switch to the interrupt subroutine pointed to by interrupt vectors in page FFH.

In normal operation, the I-flag will be set to I = 1 by operation of the 6502 reset ($\overline{\text{RES}}$) line. Thus, when power is first applied, and a power-on reset pulse generated, the I-flag will be set. If the programmer wishes to allow interrupts, then the program must clear the I-flag (i.e., reset to I = 0) using the CLI (clear interrupt) instruction. The I-flag is also reset to I = 0 by the PLP (pull processor status from stack) instruction and during an RTI (return from interrupt) *if* the I-flag was already zero prior to going to the interrupt subroutine. This latter condition is necessary because, otherwise, the program would have to re-execute the CLI instruction or be content with permitting only one interrupt.

Zero (Z) Flag. The Z-flag is used to indicate whether the result of the previous instruction was either zero or non-zero. If the result is zero (00H), then the Z-flag is 1; if the result is anything other than 00H, then the Z-flag is 0. The main use for the Z-flag is in the test-and-branch operations, most often involving the BNE (branch on result not equal zero) and BEQ (branch on result equal zero) instructions. Note that the Z-flag is *not* affected during decimal mode (D-flag = 1) additions (ADC) or subtractions (SBC). The following instructions can affect the Z-flag:

ADC	CPY	EOR	LDX	PLP	TAX
AND	CPX	INC	LDY	ROL	TAY
ASL	DEC	INX	LSR	ROR	TXA
BIT	DEX	INY	ORA	RTI	TSX
CMP	DEY	LDA	PLA	SBC	TYA

The programmer cannot directly affect the Z-flag, but there are schemes by which the programmer can indirectly cause the Z-flag to be set or reset. We can use LDA, LDX, or LDY to load one of the registers with 00H if we want Z = 1, or any other number (except 00H) if we want Z = 0. If the register is in use, then temporarily transfer the contents to a location in page zero. As with the N-flag, however, one must be careful not to affect the Z-flag when data are recovered.

Carry (C) Flag. The C-flag is used to indicate a carry or borrow situation resulting from an arithmetic operation. In some cases (the shift/rotate instructions), the C-flag becomes a ninth bit in the accumulator. Instructions affecting the C-flag are:

ADC
ASL
CLC
CMP
CPX
CPY
LSR
PLP
ROL
ROR
RTI
SBC
SEC

The carry flag can be set (C = 1) by SEC, and reset (C = 0) by CLC. Sometimes, when arithmetic or logical operations do not produce the expected result, a little investigation reveals that the C-flag had been set on a previous operation and will therefore affect the current result. In that case, a CLC is executed to clear the carry flag.

MANIPULATING PSR

Two instructions will help us manipulate the processor status register (PSR) flags: PHP and PLP. The PHP instruction pushes the contents of the PSR onto the stack indicated by the 6502 Stack Pointer (SP) register. The PLP reverses the order, and pulls the next value off the stack and places it in the PSR. The effect of PLP can be profound, especially if the stack is used again after PHP. Be careful!

7

6502 General Instruction Set

The 6502 instruction set is not as extensive as, say, the Z-80™ instruction set, but is sufficiently flexible to permit all functions expected of a microprocessor. In this chapter we will discuss the instructions generally, leaving specific details for the tables in Chapter 16.

INSTRUCTIONS

An instruction tells the computer what operation is to be performed. To the computer, these instructions are binary numbers (sometimes written as a 2-digit hexadecimal number) stored in memory. These instructions look like all other binary numbers in memory. The way the computer knows that a given number is an instruction, rather than a data word or alphanumeric character representation, is that it is fetched during an instruction fetch machine cycle. For example, suppose the 6502 encounters 69H (01101001_2) at some memory location specified by the program counter. This same pattern could be binary for the base-10 number 105_{10}, or the ASCII character i, or the instruction ADC, *immediate*. It is the job of the programmer to ensure that binary numbers at any location are necessary.

Instructions, then, are binary codes which tell the computer what to do, i.e., what operation must be carried out.

For the convenience of programmers, each instruction is given a descriptive mnemonic. When we see the mnemonic ADC, #nn, we know immediately what is meant, whereas 69H could be quite meaningless without a look-up chart of instructions sorted by op-code.

6502 INSTRUCTIONS

The 6502 instruction set is broken into three main categories: Group-I, Group-II, and Group-III. The Group-I instructions tend to be the most flexible, and have the most addressing modes. Examples of Group-I instructions include *load, add*, and *store*. The Group-I instructions include:

ADC	Add with carry
AND	Logical-AND
CMP	Compare
EOR	Exclusive-OR
LDA	Load Accumulator
ORA	Logical-OR
SBC	Subtract with borrow
STA	Store Accumulator

All of the Group-I instructions respond to the following addressing modes:

Immediate
Zero Page
Zero Page, X
Absolute
Absolute Indexed, X
Absolute Indexed, Y
Indexed Indirect
Indirect Indexed

Group-II instructions are those such as *shift, increment register, decrement register*, and the *register-X movement* instruction. Group-II is broken into two subgroups which we will call Group-IIa and Group-IIb. The instructions in Group-IIa are the *shift* and *rotate* instructions.

LSR	Shift Right
ASL	Shift Left
ROL	Rotate Left
ROR	Rotate Right

Group-IIb instructions include the following:

INC	Increment
DEC	Decrement

LDX	Load-X
STX	Store-X

The available addressing modes for Group-II instructions include:

Zero Page

Zero Page, X

Absolute

Absolute Indexed, X

Accumulator

Group-III instructions are all of those which do not fall into either Group-I or Group-II, including stack operations, Register-Y operations, and X-Y compares.

In the rest of this chapter we will consider the instructions, their operation, and the associated mnemonics. Much of the information here will be repeated in Chapter 16 where we will tabulate the information, as well as giving the op-codes in hexadecimal, binary, and octal forms.

GROUP-I INSTRUCTIONS

The Group-I instructions include ADC, AND, CMP, EOR, LDA, ORA, SBC, and STA. We will consider these instructions according to the following functional groups:

1. LOAD and STORE.
2. ARITHMETIC.
3. LOGICAL.
4. COMPARE.

Similar functional groups will be found in the Group-II and Group-III categories (for example, LDA is very similar to LDX and LDY, despite being in different groups).

Load and Store Instructions (LDA and STA)

The *load* and *store* instructions refer to data in the accumulator or A-register, and their movement to and from memory. A load instruction moves data from a memory *location* to the accumulator. The mnemonic for the load instruction is LDA, or LoaD Accumulator. When an LDA instruction is executed, the result is that some datum will be placed in the accumulator either directly or by transfer from some designated memory location. Let's step through the operation of the various LDA instructions in the various addressing modes.

LDA #nn (Immediate). This instruction uses the immediate addressing mode, and is a 2-byte instruction. The first byte will be the op-code (A 9H), while the second byte is the number which will be placed in the accumulator. Thus, when the following is encountered:

A9H
34H

the 6502 will load the accumulator with the hexadecimal number 34H (see Figure 7-1). In most assembly language formats, the above instruction would be written LDA #34.

LDA nn (Zero Page.) The zero page version of LDA is a 2-byte instruction that will operate only on locations in page zero, i.e., the 256 bytes from 0000H to 00FFH. The first byte of the instruction is the op-code (A5H), while the second byte defines the address in page zero where the data to be loaded will be found. For example, suppose the program encounters the following instruction:

A5H
52H

The op-code A5H tells the 6502 to load the accumulator with page zero data found at location 0052H. Figure 7-2 shows the operation of this instruction. The action is:

1. While executing the program the LDA (52H) instruction is encountered.
2. The 6502 goes to memory location 0052H, where it finds data 67H.

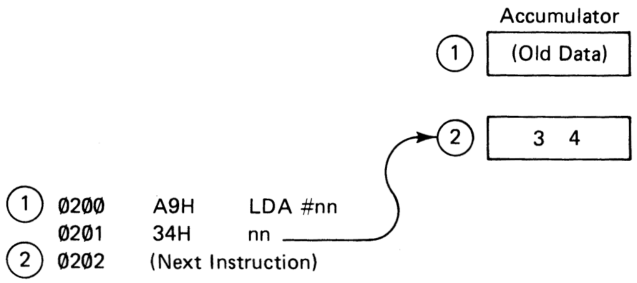


Figure 7-1. LOAD-immediate (LDA #nn) instruction

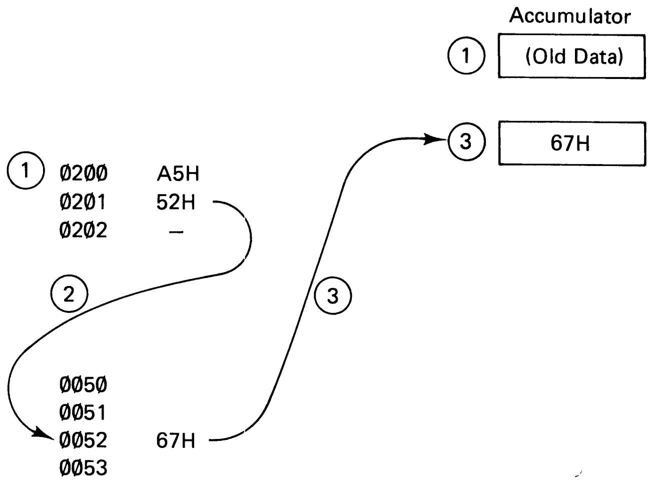


Figure 7-2. LOAD-zero page instruction

3. The data from 0052H (i.e., the number 67H) is loaded into the accumulator.

LDA nn,X (Zero Page, X). This instruction has a bit more flexibility than simple zero page addressing. For the LDA nn,X instruction, the effective address of the page zero address where the data are found is computed by adding the contents of the X-register in the 6502 to the second byte of the instruction.

For example, assume that the X-register contains 03H when the following instruction is encountered:

Byte 1	B5H	(LDA nn,X)
Byte 2	50H	nn

The 6502 will compute the zero page address by adding byte 2 to the contents of the X-register:

nn + x =

50H + 03H

53H

Thus, the 6502 will load into the accumulator the data stored in zero page loation 0053H.

LDA nnnn (Absolute Addressing). The 3-byte absolute LDA loads the accumulator with the data stored in the two bytes that follow the op-code.

Byte 1	ADH	(LDA,nnnn)
Byte 2	Low-order address byte	
Byte 3	High-order address byte	

Suppose we encounter the code:

Byte 1	ADH	op-code LDA
Byte 2	53H	nn
Byte 3	0FH	nn

The 6502 will load the accumulator with the contents of the memory location specified by (byte 3) + (byte 2), which in this case is 0F53H.

LDA nnnn,X (Absolute,X). The absolute-X mode LDA instruction uses three consecutive bytes to designate an address defined by the sum of the contents of the X-register and the absolute address given in bytes 2 and 3 of the code. For example, assume that the X-register contains the number 05H when the following code is encountered during program execution:

Byte 1	BDH	LDA nnnn,X
Byte 2	00H	nn
Byte 3	0EH	nn

The absolute address is defined by (byte 3) + (byte 2), so is 0E00H. Since BD is the hexadecimal op-code for LDA nnnn,X, the actual address is 0E00H + X, or 0E00H + 05H, which is 0E05H.

The LDA nnnn,X instruction is particularly useful for accessing look-up tables. In the example here, we could store a 256-element table from 0E00H to 0EFFH, and either step through the table or access specific data by manipulating the data in the X-register. This type of strategy is used in programs such as code conversion (e.g., ASCII to Baudot) or in the linearization of transducer or instrumentation data.

LDA nnnn,Y (Absolute Y). The Absolute-Y LDA instruction is the same as the Absolute-X LDA instruction, except that the effective address is computed by adding the contents of the Y-register to the absolute address specified by the second and third bytes of the code.

$$\text{Address} = (\text{Byte 3} + \text{Byte 2}) + Y$$

For example if the contents of the Y-register in the 6502 are EAH, and the following code is encountered:

Byte 1	B9H	LDA nnnn, Y
Byte 2	00H	nn
Byte 3	02H	nn

the effective address is computed as 0200H + EAH, or 02EAH. The uses of the Absolute-Y LDA instruction are the same as the Absolute-X LDA instruction.

LDA (nn,X) Indirect, X. The *indirect indexed* LDA instruction combines the indirect and indexed addressing modes in a technique called pre-indexing. The effective address of the data to be stored in the accumulator is stored in two successive locations in page zero. An *indirect indexed* instruction is a 2-byte instruction and uses the X-register. The page zero address containing the low-order byte of the effective address is computed by adding the second byte of the instruction to the contents of the X-register.

Figure 7-3 shows an example of how the *Indirect, X* instruction

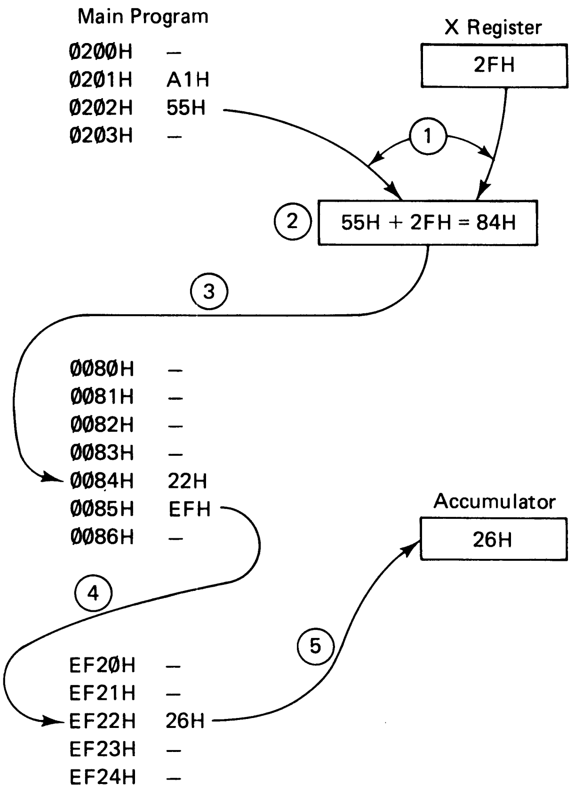


Figure 7-3. Operation of an Indirect-X instruction (pre-indexing technique)

operates. The contents of the X-register are 2FH, and the 6502 is executing a program in page two when, at 0201H, it encounters:

Byte 1	A1H	LDA (nn,X)
Byte 2	55H	nn

The 6502 interprets "A1H" as the LDA (nn,X) instruction, so the following sequence (see Figure 7-3) takes place:

1. The LDA (55,X) instruction is fetched and decoded.
2. The 6502 adds together byte 2 of the instruction (55H) and the contents of the X-register (2FH) to obtain a result of:

   55H + 2FH = 84H

3. Step 2 tells the 6502 that the low-order byte of the effective address of the data to be loaded into the accumulator will be found at location 0084H in page zero; the high-order byte of the effective address is stored at the next sequential location (0084H + 01H), which is 0085H.
4. The address stored at 0084H and 0085H is EF22H, so the program counter of the 6502 will be loaded with EF22H.
5. The contents of memory location EF22H (26H) are stored in the accumulator.

LDA (nn), Y Indirect, Y. The *Indirect, Y LDA* instruction is similar to *Indirect, X* but uses post-indexing rather than pre-indexing. Whereas pre-indexed addressing involves indexed indirect addressing, the post-indexing method uses indirect indexed addressing. Sound confusing? Well, Figure 7-4 may help a little. You may wish to reexamine Figure 7-3 after you read the description below, and compare these two similar LDA modes. The Y-register contains the hexadecimal number ACH, and the main program is executing instructions in page two when the following is encountered:

Byte 1	0201H	B1H	LDA (nn), Y
Byte 2	0202H	4CH	nn

The 6502 interprets this code as an LDA (4CH), Y instruction, so the following operations take place:

1. The LDA (4CH), Y instruction is encountered and decoded, telling the 6502 that the indirect address is stored in locations 004CH (low-order byte) and 004DH (high-order byte) of page zero.

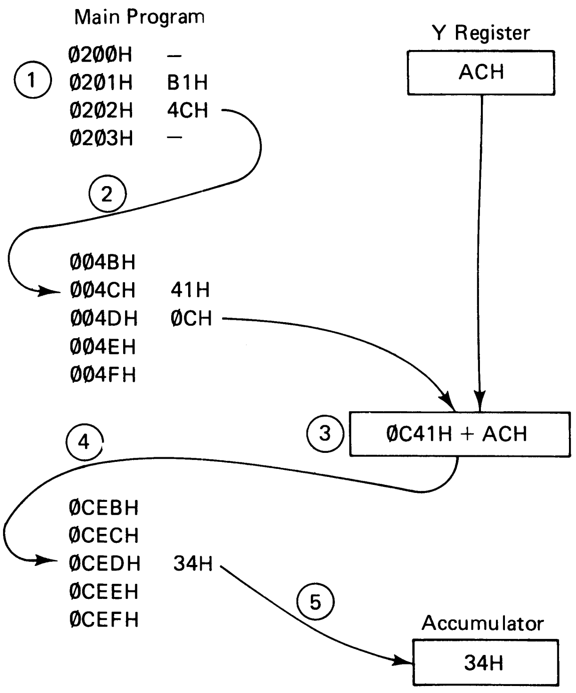


Figure 7-4. Operation of an Indirect-Y instruction (post-indexing technique)

2. The 6502 goes to 004CH and 004DH and finds address 0C41H. This is not the effective address, but must be added to the contents of the Y-register.
3. The indirect address (0C41H) is added to the contents of the Y-register (ACH) to yield the actual effective address:

$$0C41H + ACH = 0CEDH$$

4. The address 0CEDH is loaded into the 6502 program counter.
5. The contents of memory location 0CEDH (34H) are loaded into the accumulator.

Summary of LDA. All versions of the LDA instruction have the effect of fetching data from some point in memory and storing it in the accumulator of the 6502. In some cases, discovering the location of the actual data is complex (as in Indirect,X or Y), while in others it is very simple, e.g., in the LDA, Immediate instruction. In all cases, however, the end result is that data from some specified or computed location in memory wind up in the accumulator.

Note: This data transfer is nondestructive! If we execute an LDA nnnn (Absolute) instruction, we will *copy* the data at the location specified by nnnn into the accumulator. Following execution of this instruction, the same data will appear at *both* locations, i.e., nnnn and the accumulator. Figure 7-5 shows the situation for both pre- and post-execution of an instruction.

The STA instructions are exactly the opposite of LDA. Whereas the LDA instruction will cause data to be loaded into the accumulator, the STA causes data to be copied from the accumulator to some specified location in memory. Once again, we find the operation is non-destructive. In other words, if an instruction causes data to be transferred from the accumulator to some memory location, then after

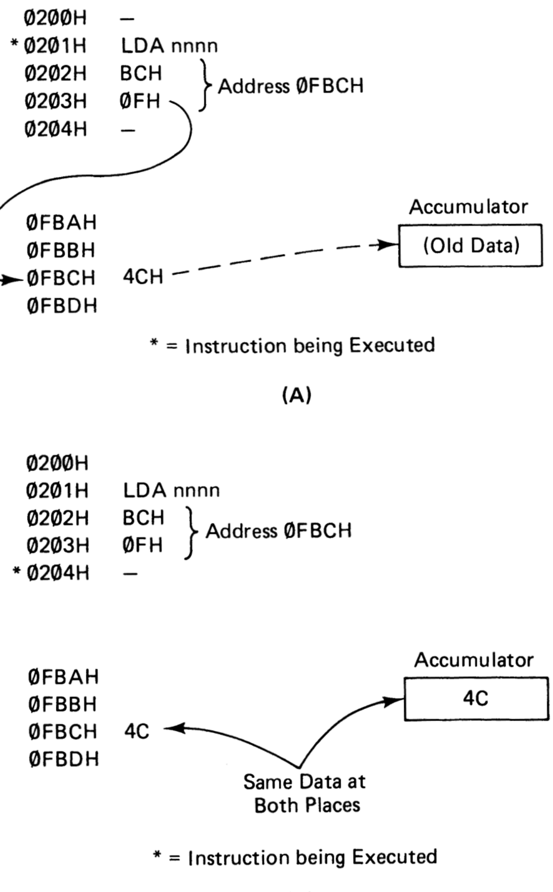


Figure 7-5. Status A) pre-execution and B) post-execution shows the non-destructive nature of data transfer

execution of this instruction the data will appear in both the accumulator and the designated memory location.

The STA instructions use all of the addressing modes of the LDA, except for the immediate addressing mode (which would be illogical for STA since "STA" stands for *Store Accumulator!*). It will serve little purpose to reiterate the lengthy descriptions of instruction action as given above for LDA, because the only difference is the direction of data transfer with respect to the accumulator.

The STA and LDA instructions are frequently used together, especially in computer I/O operations. For example, one popular 6502-based microcomputer memory maps an input port at location A001H. Suppose we want to input this data and then save it by storing it in some location in memory. This may be necessary (in fact, it usually is!) because some subsequent instruction may alter the contents of the accumulator where the input data from the port is at A001H and store it at location EF05H. A typical program fragment to accomplish this trick would be:

Mnemonic	*Code*	*Comment*
LDA (A001)	ADH	Load accumulator with contents
nn-low	01H	of location A001H
nn-high	A0H	
STA (EF05H)	8DH	Store contents of accumulator at
nn-low	05H	EF0SH
nn-high	EFH	

We might also use STA and LDA in conjunction with each other to temporarily store accumulator data which will be used again. A brief example is:

1. LDA (A001H0).
2. STA (0050H).
3. (Other programming).
4. LDA (0050H).
5. (Other programming using retrieved data).

In step 1 we loaded data into the accumulator from our input port at location A001H. During step 2 that data was temporarily stored at location 0050H in page zero. Step 3 has the 6502 doing other things for awhile, a phase which might take from one to any number of instructions. Step 4 has the data stored in 0050H retrieved by reloading it into the accumulator. Step 5 shows the program continuing, using the retrieved data.

The STA instructions affect none of the PSR flags. However, the LDA affects the N-flag to indicate whether the loaded data are negative (N = HIGH) or positive (N = LOW). The Z-flag is also affected by LDA and indicates whether the loaded data is zero (Z = HIGH) or non-zero (z = LOW).

Arithmetic Instructions (ADC and SBC)

The arithmetic instructions form another subset of the Group-I instruction block. Computers are really dumb devices because all they can do is add and subtract. Even when subtracting, the computer is really using addition, but fools the computer into thinking it's adding instead of subtracting by making the subtrahend a two's complement equivalent of the number being subtracted. Multiplication and division are handled using either software algorithms or specialized external hardware. The 6502 has the *add-with-carry* (ADC) and *subtract-with-carry* (SBC) instructions.

The ADC instruction will add the contents of the accumulator to data specified by the instruction. All eight Group-I addressing modes are allowable for ADC. Let's consider the rules for binary arithmetic:

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 0 \text{ Carry-1} \end{aligned}$$

A simple example follows: Add the binary numbers 10000101_2 (i.e., 85H) and 10011001_2 (i.e., 99H):

$$\begin{array}{r r} & 1 \\ & 1\ 0\ 0\ 0\ 0\ 1\ 0\ 1 \\ & \underline{1\ 0\ 0\ 1\ 1\ 0\ 0\ 1} \\ \text{(Carry-1)} & 0\ 0\ 0\ 1\ 1\ 1\ 1\ 0 \end{array}$$

The answer is 00011110_2 (IEH) plus a carry-1. On the 6502, the accumulator would contain 1EH and the carry flag (C) would be set to equal HIGH or 1 following this operation. Note: if you have a hexadecimal calculator, such as the TI Programmer, the display will read "11E" (hex).

The immediate addressing mode is a 2-byte instruction that will add the contents of the accumulator with the data in the second byte of the instruction, and then store the result in the accumulator. For example, suppose the accumulator contains 3FH when the following instruction is encountered:

Byte 1	69H	ADC #nn
Byte 2	24H	nn

Since 69H is the op-code for the *ADC, immediate* instruction, and the accumulator contents are 3FH, the following addition takes place:

3FH	(accumulator data)
+24H	(Byte 2 data)
63H	(answer stored in accumulator)

In the above example, the carry flag would be reset (C = LOW).

The other seven addressing modes allowed the ADC instruction will add to the contents of the accumulator data retrieved from memory in the manner defined by the protocol for the specific addressing mode.

The symbolic notation for the ADC instruction is:

$$A + M + C \qquad A$$

which means, "The contents of the accumulator are added to data retrieved from memory (M) and the carry flag (C)." The carry flag may be set prior to the addition operation, and remains set when new instructions are encountered. Let's look at an example of how this could affect an addition problem. If we add 5FH and 42H, the answer should be A1H. But suppose the carry flag had been previously set (C = 1) by another operation? Although you might believe that the problem being worked is:

$$5FH + 42H = A1H$$

the *actual* problem is:

$$5FH + 42H = 01H = A2H$$

The solution will be in error because the programmer failed to account for the 01H represented by the carry flag. The answer is to clear the carry flag (i.e., make C = 0) prior to the addition. For example, suppose we do not want the C-flag to affect an addition such as above, we could write the following (assume 5FH is the accumulator data):

CLC	Clear carry flag
ADC #42H	Add #42H to accumulator data

Since the carry flag was cleared prior to the ADC #42H instruction, the result will be the desired A1H.

The *subtraction with carry* (SBC) instruction is actually a subtraction with BORROW, if we use mathematically correct terminology. The symbolic operation for SBC is:

$$A - M - \overline{C} \rightarrow A$$

This notation says that the value fetched from memory (M) and the complement of the carry flag ($\overline{C}$) is subtracted from the contents of the accumulator, and the result is stored in the accumulator. Note that the carry flag will be *set* (HIGH) if a result is equal to or greater than zero, and *reset* (LOW) if the results are less than zero, i.e., negative.

The SBC instruction has all eight Group-I addressing modes available, as was also true of ADC.

The SBC instruction affects the following PSR flags: negative (N), zero (Z), carry (C), and overflow (V). The N-flag indicates a negative result and will be HIGH; the Z-flag is HIGH if the result of the SBC instruction is zero and LOW otherwise; the overflow flag (V) is HIGH when the result exceeds the values 7FH ($+127_{10}$) and 80H with C = 1 (i.e., -128_{10}).

The 6502 manufacturer recommends for single-precision (i.e., 8 bit) subtracts that the programmer ensure that the carry flag is set prior to the SBC operation to be sure that true two's complement arithmetic takes place. We can set the carry flag by executing the SEC (*set carry flag*) instruction.

The rules for binary subtraction are:

$0 - 0 = 0$	
$0 - 1 = 0$	Carry-1
$1 - 0 = 1$	
$1 - 1 = 0$	

The SBC instruction complements the ADC instruction and is used in arithmetic operations. The one additional instruction used in arithmetic operations is the *set decimal mode* instruction that permits binary coded decimal (BCD) arithmetic. But since it is not a Group-I instruction, it will be discussed elsewhere.

Logical Instructions (AND, ORA, and EOR)

The 6502 microprocessor can perform three logical functions: AND, OR (ORA), and the exclusive-OR (EOR). The 6502, as with other microprocessor chips, performs these operations on multibit binary words on a bit-by-bit basis. In other words, the results of a logical operation on one pair of bits (e.g., b0 and a0) will not affect operations on the next higher (e.g., b1 and a1) or lower order bit.

The logical-AND operation obeys the following rules:

$$0 \text{ AND } 0 = 0$$
$$0 \text{ AND } 1 = 0$$
$$1 \text{ AND } 0 = 0$$
$$1 \text{ AND } 1 = 1$$

A fact worth remembering for the AND operation is that the result is always LOW (0) unless both bits are HIGH (1). We use this fact in bit making operations. For example, we often tell whether or not a 7-bit ASCII keyboard is sending new data by applying the strobe bit to bit 7 of an input port. We could mask all bits except bit 7 and test for non-zero. The ASCII for the character "M" is 4 DH, which in 7-bit binary notation is 1001101_2, or if a strobe (data valid) bit is added at bit 7, the code becomes 11001101_2 (CDH). To test this data for validity, we could AND CDH with 80H. Here it is shown in binary to illustrate the principle:

```
     1 1 0 0 1 1 0 1
AND  1 0 0 0 0 0 0 0
     ---------------
     1 0 0 0 0 0 0 0
```

This is for the data valid condition—the result is non-zero and that is testable, or, for the data-not-valid condition, when bit 7 is LOW:

```
     0 1 0 0 1 01001
AND  1 0 0 0 0 0 0 0
     ---------------
     0 0 0 0 0 0 0 0
```

the result is zero.

The 6502 AND instruction performs on a bit-by-bit basis, stores the result in the accumulator, and enjoys all eight Group-I addressing modes. The AND instruction affects the N-flag and Z-flag.

The logical-OR instruction (ORA) is the complement of the AND instruction. Whereas the result of the AND instruction was true (1) only when both bits are true, the OR will be true when either or both bits are true:

$$0 \text{ OR } 0 = 0$$
$$0 \text{ OR } 1 = 1$$
$$1 \text{ OR } 0 = 1$$
$$1 \text{ OR } 1 = 1$$

Again, the operation is performed on a bit-by-bit basis in the 6502, so no operation between bits of any order will affect operation of ORA command and on any other set of bits.

The ORA instruction affects the N-flag and Z-flag. The N-flag will be HIGH if bit 7 of the result is HIGH, and low otherwise. The Z-flag will be HIGH if the result is zero, and LOW if the result is non-zero.

Exclusive-OR (EOR) instruction is interesting. The result is true (1) if either bit is true, but not if both bits are true. The rules for the exclusive-OR are:

$$0 \text{ XOR } 0 = 0$$
$$0 \text{ XOR } 1 = 1$$
$$1 \text{ XOR } 0 = 1$$
$$1 \text{ XOR } 1 = 0$$

Note that any time the two bits are the same (both 0 or both 1), the result will be 0. The Logical Exclusive-OR function is called "XOR" in digital electronics texts, but the 6502 Exclusive-OR instruction is EOR.

The EOR instruction can use all eight Group-I addressing modes, and will affect the N-flag and Z-flag.

EOR is used in arithmetic operations and others, but one use is *complementing* the accumulator. This is done by using the EOR instruction in the immediate addressing mode will all one's; for example, B1H XOR FFH (using binary notation for illustration):

```
    1 0 1 1 0 0 0 1
XOR 1 1 1 1 1 1 1 1
-------------------
    0 1 0 0 1 1 1 0
```

Some single-board computers used as OEM boards, or for industrial control applications, use inverters on the input and output ports—a design feature considered ill-advised by some engineers. On those machines, we need to use the EOR #FF instruction to complement data on all I/O operations.

Compare Instructions (CMP)

The *compare* (CMP) instruction compares data fetched from memory with data stored in the accumulator *without altering the data in the accumulator.* CMP can use all eight Group-I addressing modes, and three of the PSR flags: C, N, and Z. The use of the flags is different for this instruction than for others, and operates as follows:

1. *C-flag* is *set* HIGH (1) when the value in memory is *less* than the value in the accumulator, and is *reset* LOW (0) when the value in memory is *greater* than the value in the accumulator.
2. *N-flag* is *set* HIGH (1) or *reset* LOW (0) according to the result of bit 7.
3. Z-flag is *set* HIGH (1) on *equal* comparison, *reset* for unequal comparison.

The *compare* instruction can be used for several applications, but one quoted in most of the textbooks determines which peripheral is using the interrupt capability of the 6502 to gain the attention of the processor. We can have each peripheral input a unique code, and then have the interrupt subroutine compare this code in the accumulator with several constants. By monitoring the Z-flag for "equal comparison" status, we can tell which device demands service.

GROUP-II INSTRUCTIONS

The Group-II instructions are used primarily for data manipulation and arithmetic applications. This group contains the *decrement, increment, rotate, shift,* and the *load/store* instructions for the X-register. Group-II is broken into two subgroups called Group-IIa and IIb. The former group contains the *shift* and *rotate* instructions, while the latter contains the *increment, decrement,* plus *load/store register-X* instructions.

Certain Group-II instructions use the so-called "accumulator" addressing mode in which the data used for the operand are the accumulator data. The "accumulator" addressing mode is, therefore, a special case of implied addressing.

The shift instructions are used to shift data in the accumulator either to the left (ASL) or right (LSR). Both forms of shift instruction use the following addressing modes: accumulator, zero page, zero page X, absolute, and absolute X.

The *arithmetic shift left* (ASL) instruction will shift the data in either the accumulator or the indicated memory location one position left every time it is executed; bit 7 will be transferred to the carry flag, and a 0 is stored in bit 0. This operation is shown pictorially in Figure 7-6A.

An example using the accumulator addressing mode is shown in Figures 7-6B and 7-6C. The initial condition is shown in Figure 7-6B. The data word D3H (1101001_2) is stored in the accumulator, and the state of the C-flag is irrelevant. Following execution of the ASL instruction, a 0 has been entered into the bit 0 position, and bit 7 has been moved to the C-flag. The accumulator data is not A6H (10100110_2), and the carry flag is set.

The *branch on carry clear* and *branch on carry set* instructions in Group-III can be used to alter program direction after each shift according to the condition of the C-flag. These branch instructions will be discussed with other Group-III instructions.

In addition to the C-flag (which always takes on the previous value of bit 7), the ASL instruction also affects the Z- and N-flags. The Z-flag is set (1 or HIGH) if the result of the shift produces a zero result. While this condition could occur at any time if the correct data were

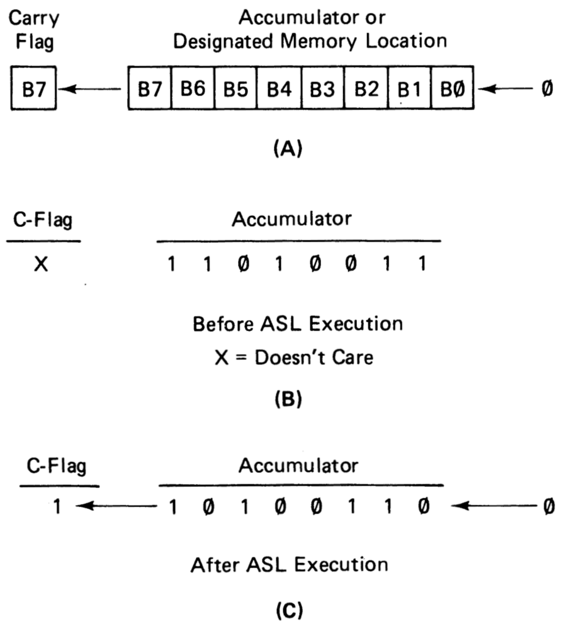


Figure 7-6. Operation of the Arithmetic Shift Left (ASL) instruction A) operation, B) status before execution, C) status after execution

present, it will always occur on the eighth shift of the same data because we have been entering zeros into bit 0 each time execution occurs. After this operation occurs eight times, all bits will be zero. The N-flag will take on a value that is determined by the condition of bit 7 following execution. Since bit 6 is shifted to the bit 7 position, the N-flag is set according to the previous value of bit 6. If the result bit 7 is 1, then N = 1; if result bit-7 is 0, then N = 0.

Accumulator mode instructions operate on data in the accumulator, while the other addressing modes will modify memory location data *without* affecting other registers in the 6502.

The *logical shift right*, or LSR, instruction is similar to, but exactly the opposite of, ASL. The LSR instruction shifts data to the right, rather than the left. In execution of LSR, bit 0 is stored in the C-flag and a zero is entered into the bit 7 position.

Two principal uses for ASL and LSR instructions are in multiplication/division arithmetic operations, a parallel-to-serial data conversion (serial-to-parallel conversion is also possible, but is more involved). We gain the arithmetic capability because each left shift (ASL) will multiply the data by two, while each right shift divides the data by two.

The *rotate left* (ROL) and *rotate right* (ROR) instructions are similar to the *shift* instructions, except that data are recirculated back into the accumulator or memory location addressed by the instruction. In both cases, data are shifted one bit position left or right according to which *rotate* instruction is being executed. The difference between *rotate* and *shift* instructions is illustrated by the following:

1. ROR (*rotate right*). Each bit is shifted one bit to the right, the contents of the C-flag are shifted into bit 7, and bit 0 is shifted to the C-flag. Thus, after nine shifts, the contents of the accumulator (or designated memory location) will be exactly the same as before, as will be the C-flag.
2. The ROL (*rotate left*) instruction works exactly the opposite of ROR: bit 7 goes to the C-flag and the C-flag goes to bit 0.

The *rotate* instructions have uses similar to the *shift* instructions, but the data can be recirculated back into the accumulator (or memory location); thus the operation is nondestructive of data.

The Group-IIb instructions are: *Increment* (INC), *Decrement* (DEC), *load X* (LDX), and *store X* (STX). The DEC and INC instructions are used for addressing modes: zero page, zero page X, Absolute, and Absolute X. The LDX and STC instructions recognize these same four addressing modes plus the immediate addressing mode.

The INC affects a designated memory location, and will increase the value of the data word at that location by 1. In other words, the operation is:

$$M + 1 \rightarrow M$$

The increment INC instruction does not affect the accumulator, but does affect the N and Z-flags. The N-flag will be 1 when bit 7 of the result after execution of INC is 1, and 0 if bit 7 is 0. The Z-flag will be 1 when the result is zero (e.g., where FFH + 1 = 00H), and 0 if the result is non-zero.

The DEC instruction is exactly the opposite of the INC instruction. DEC causes the contents of the designated memory location to be reduced by one, or symbolically:

$$M - 1 \rightarrow M$$

The N and Z-flags are affected in exactly the same manner as in INC.

The LDX (load X-register) and STX (store X-register contents in a designated memory location) are analogous to LDA and STA discussed earlier. The LDX uses the following addressing modes: immediate, zero page, zero page X, Absolute, and Absolute X. Symbolically, the operation is:

$$M \rightarrow X$$

In other words, a data word from memory is loaded into the 6502 internal X-register. The N and Z-flags are set according to the result, i.e., the value of the data word stored in the X-register.

The STX (store X) instruction has the effect of storing the contents of the X-register at a designated memory location. Only the zero page, zero page Y, and Absolute addressing modes are permitted this instruction. No PSR flags are affected by the STX instruction.

Thus far we have discussed the Group-I and Group-II instructions. The Group-III instructions include all other instructions in the 6502 repertoire.

GROUP-III INSTRUCTIONS

The Group-III instructions include the following:

BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result equal to Zero

BIT	Bit Test
BMI	Branch on Result Minus
BNE	Branch on Result not equal Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Flag
CLV	Clear Overflow Flag
CPX	Compare Memory with X-register
CPY	Compare Memory with Y-register
DEX	Decrement X-register
DEY	Decrement Y-register
INX	Increment X-register
INY	Increment Y-register
JMP	Jump to New Memory Location
JSR	Jump to Subroutine
LDY	Load Y-register
NOP	No Operation
PHA	Push Accumulator on Stack
PHP	Push Processor Status from Stack
RTI	Return from Interrupt
RTS	Return from Subroutine
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable
STY	Store Y-register in Memory
TAX	Transfer Accumulator to X-register
TAY	Transfer Accumulator to Y-register
TYA	Transfer Y-register to Accumulator
TSX	Transfer Stack Pointer to X-register
TXA	Transfer X-register to Accumulator
TXS	Transfer X-register to Stack Pointer

Let's briefly describe these instructions and their operation. Chapter 16 contains op-codes and other details on instructions.

BCC (Branch on Carry Clear). This instruction uses relative addressing to branch forward and backward in the program if the carry flag is clear (0). BCC doesn't affect the PSR flags.

This instruction is the first of several branch instructions which we will consider. Since all use similar protocols regarding the direction and distance of the relative addressed branch, we will cover BCC in detail but delete the detail in discussion of the other instructions. Previously, we did the same thing by covering all eight addressing modes for LDA, but not the Group-I instructions which followed.

The BCC instruction tests the carry flag in the Processor Status Register (PSR) of the 6502. The C-flag can have only two states, set (1) or clear (0). If the C-flag is set, then the BCC instruction will not cause a branch. The program is said to "fall through" the BCC, which is jargon for the program will execute the next instruction in sequence, rather than branching.

BCC is a 2-byte instruction with the mnemonic form "BCCnn," where "nn" will specify the direction and distance of the branch. This instruction will be encountered in the form:

Byte 1	90H	BCC	nn
Byte 2	nn		

Relative addressing means that the program will branch or "jump" to a location relative to the instruction. The 6502 will permit

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

(A)

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

(B)

Figure 7-7. A) forward branch table, B) backward branch table

branches of up to 127 spaces forward ($+127_{10}$), or 128 spaces backward (-128_{10}) as counted from the next instruction following BCC. Forward branches will use values for nn that are positive hexadecimal numbers, while backward branches will use the hexadecimal notation for the two's complement representation of the negative number. Forward and backward hexadecimal codes are given in Figures 7-7A and 7-7B, respectively.

Let's look at three examples: a no-branch, a forward branch of $+6_{10}$, and a backward branch of -6_{10}. Figure 7-8 shows the no-branch condition.

Recall that BCC is *branch on carry clear* (C = 0). At location E003H the program encounters BCC 06H, so it goes to the C-flag to determine whether it is set (1) or reset (0). In this case, C = 1, so no branch will occur.

The program "falls through" to execute the instruction at E005H. The no-branch conditon may be the trivial case, but essentially it permits continued execution of a program unless some specified criterium (e.g., C = 0) is met.

An example of a *forward branch BCC* instruction is shown in Figure 7-9. In this case the instruction is BCC 06H, which means that the program will jump six steps forward when the carry flag is zero.

A fundamental error made by many beginning programmers involves the counting of the six steps. Counting begins at the location following the second byte of the instruction, since the instruction is at E003H and E005H, in the branch condition, the next instruction will be at E004H+06H, or E00AH. In this example, the BCC 06H instruction is encountered at E003H, so the 6502 checks the C-flag, finds that it is zero, and jumps six steps ahead to fetch the next instruction at E00AH.

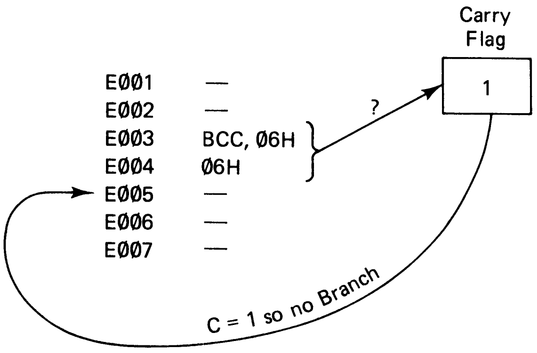


Figure 7-8. Operation for the NO BRANCH condition

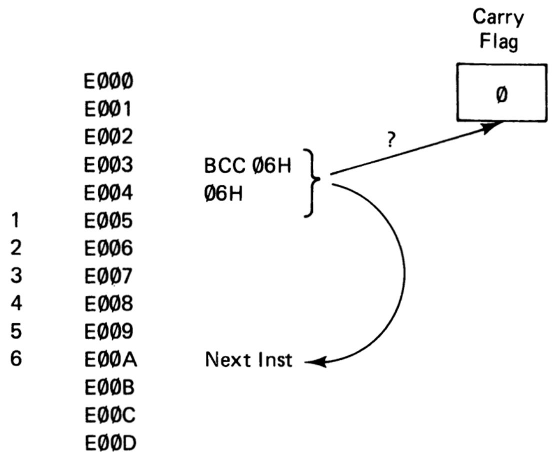


Figure 7-9. Forward branch

Figure 7-10 shows an example of a backward branch version of the BCC instruction. The relative branching distance is still six, but in this case it is −6. The two's complement notation for −6 in hexadecimal form is FAH, so the second byte will be FAH. Once again, the counting takes place from the location following the second byte. Since in Figure 7-10 the BCC FAH instruction is located at E008H and E000H, E00AH is the point where counting is referenced. Thus, six back from E00AH will be E004H. When the program encounters BCC FAH (BCC −06) at E008H, the 6502 tests the C-flag, finds it zero and transfers program control to E004H.

Of course, in both forward and backward branch cases, the change in program control is implemented within the 6502 by altering the contents of the program counter register.

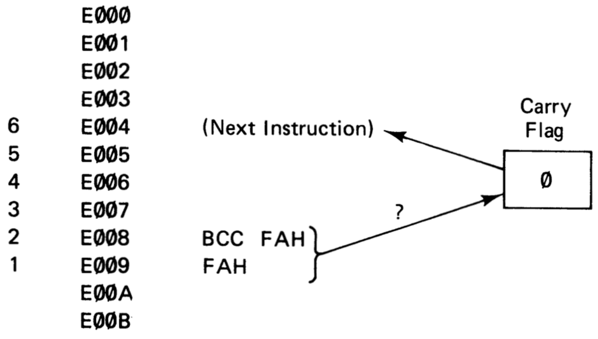


Figure 7-10. Backward branch

The -127_{10} to $+128_{10}$ locations' range for the relative branch can be increased by having the instruction at the branch-to location be at jump instruction; for example.

```
E001     BCC     06H
E002     06H
E003
E004
E005
E006
E007
E008     JMP     F008H
E009     08H
E00A     F0H
  .
  .
  .
F008H
```

In the case above, BCC 06H branches to E008H, where a JMP F008H is found. Thus, this BCC 06H instruction causes a much larger branch, i.e., to F008H. The use of a JMP instruction, then, can make the range of any conditional branch instruction equal to the available memory.

There is a bit of confusion among some new programmers regarding the relative branch distance. In some cases, the distance is listed as +129 and −126, while in others, the +127 and −128 figures are listed. The difference is merely a matter of where one starts measuring. The 129/126 protocol is from the current program counter contents, while the 127/128 figure is derived from counting from the next instruction following the conditional branch instruction. The difference is due to the 2-byte instruction. If you add to lower figures you get the other figures:

$$(+127) + (2) = +129$$

$$(-126) + (-2) = -128$$

We will not discuss the rest of the conditional branch instructions in the detail of the BCC instruction because the branching protocols are the same. The branch conditions and flags affected will be discussed briefly.

BCS (Branch on Carry Set). This instruction is the exact inverse of the BCC instruction. Branching occurs when the carry flag is set (1), rather than reset. No flags are affected by the BCC instruction. The program counter (PC) will be affected only if C = 1. Addressing mode is relative.

BEQ (Branch on Result Equal Zero). Branch occurs if the result of an operation is zero, i.e., if the zero flag is set (Z = 1). No flags are affected, and PC is affected only for Z = 1 (i.e., result is zero). Addressing mode is relative.

BIT (Bit-Test between Memory and Accumulator). A logical-AND is performed between the contents of the accumulator and the contents of a specified memory location. The result of the comparison is *not* stored in the accumulator, so the accumulator contents remain unaffected. The contents of the Processor Status Register (PSR) are affected as follows:

1. N-flag is set to the value of bit 7 of the data in the selected location.
2. V-flag is set to the value of bit 6 of the data in the selected memory location.
3. Z-flag is set (Z = 1) if the result of the logical-AND is zero, and reset (Z = 0) if the result is non-zero.

The BIT instruction uses only the zero page and Absolute addressing modes.

The BIT instruction performs a logical-AND between the accumulator and the contents of the designated memory location on a bit-by-bit basis.

Recall that the result of a logical-AND will be 1 only if both bits are 1:

0 AND 0 = 0

0 AND 1 = 0

1 AND 0 = 0

1 AND 1 = 1

Thus, we can test any of the 8 bits by manipulation of the data in the memory location. For example, suppose we want to test bit 4 of the accumulator data. Suppose the accumulator data is 93H (10010011_2) and we want to detect a 1 in bit 4. The operation of BIT is shown in Figure 7-11. In order to determine if bit 4 is 1, we AND 10H from 00f1H with the contents of the accumulator. Since, in this case, the tested bit (b4) is 1, the result is non-zero, so the Z-flag is 0.

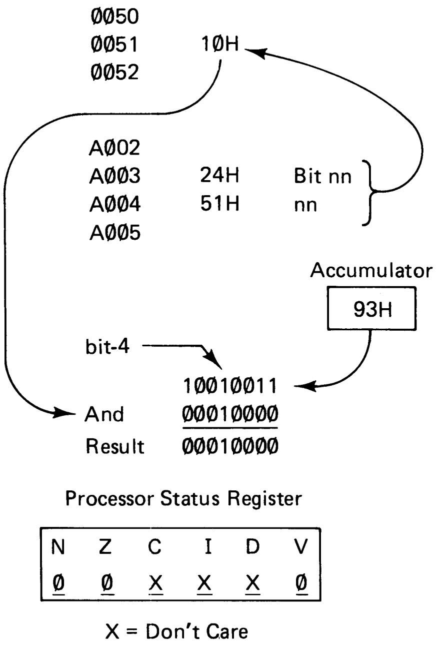


Figure 7-11. Operation of the BIT instruction

If bit 4 had been zero, then the result would have been zero, and the Z-flag would be set (Z = 1). Depending upon the desired result, a BEQ (branch on equal zero) instruction or a BNE (branch on not equal zero) instruction could be used to take action, depending on the result of BIT.

BMI (Branch on Result Minus). This instruction causes a branch operation if the result of an operation is minus, as indicated by the N-flag being set (N = 1). The N-flag will be 1 when the result bit 7 is 1. No flags are affected, but the program counter will be affected if N = 1. The relative addressing mode is used. See BCC for other operational details and examples.

BNE (Branch on Result Not Equal to Zero). This instruction is the complement of BEQ. The branch will occur if the result of an operation is non-zero, as indicated by the Z-flag being reset (Z = 0). No flags are affected, but the program counter will be affected if Z = 0. The relative addressing mode is used. See BCC for other operational details and examples.

BPL (Branch on Result Plus). This instruction is the complement of BMI. The branch occurs if the result of an operation is positive, as indicated by the N-flag being zero (N = 0). No flags are affected if N = 0; N-flag is zero when bit 7 of the result is zero. The relative addressing mode is used. See BCC for other operational details and an example.

BRK (Force Break). This instruction is a software interrupt command. When a BRK instruction is encountered, the address, the next instruction, and the contents of the Processor Status Register are pushed onto the external stack. The rectors for the BRK command are stored at FFFEH and FFFFH as follows:

FFFEH	low-order address byte
FFFFH	high-order address byte

Notice that BRK uses the same rector location as the hardware interrupt ($\overline{\text{IRQ}}$) line. The sole difference that distinguishes the BRK command from a hardware interrupt is that the B-flag is set (B = 1) if the interrupt is due to a BRK command, and reset (B = 0) if $\overline{\text{IRQ}}$ is brought LOW (0). An interrupt subroutine must contain a brief subroutine that pulls the previous PSR contents from the external stack and then tests it for B = 1 (bit 4 of the PSR) using either the AND or BIT instructions; ANDing with 10H will do the trick.

BVC (Overflow Flag Clear). This instruction is a conditional branch that will branch using relative addressing if the overflow (V) flag is clear, i.e., zero (V = 0). No flags are affected, and the program counter is affected only if V = 0. See BCC for other operational details and examples.

BVS (Overflow Flag Clear). The BVS instruction is the complement of BVC, and will branch if the overflow flag is set (V = 1). The relative addressing mode is used. No flags are affected, but the program counter will be altered if V = 1. See BCC for other operational details and examples.

CLC (Clear Carry Flag). The CLC instruction uses implied addressing and has the effect of setting the C-flag in the PSR to zero (C = 0). No other flags are affected.

CLD (Clear Decimal Mode). The CLD flag has the effect of setting the D-flag of the PSR to zero (D = 0). Implied addressing mode is used. Following execution of CLD, all subsequent ADC and SBC arithmetic operations will take place in straight binary. No flags other than the D-flag are affected.

CLI (Clear Interrupt Disable Flag). This 1-byte instruction clears the interrupt disable, or I-flag, of the PSR. Execution of this instruction causes the I-flag to become zero (I = 0). The implied addressing mode is used. The purpose of the CLI instruction is to permit the 6502 to respond to interrupt requests on the $\overline{IRQ}$ line. The I-flag is normally set to I = 1 when the 6502 is first turned on and $\overline{RST}$ is activated. The programmer must inset a CLI instruction somewhere in the program before interrupts are permitted.

CLV (Clear Overflow Flag). The CLV instruction is used to set the overflow (V) flag to zero, (V = 0). Implied addressing is used for this 1-byte instruction. No other PSR flags are affected.

CPX (Compare Memory with X-register). The CPX instruction compares the contents of the X-register with the contents of a designated memory location. Immediate, zero page, and Absolute addressing modes are used. The N, Z, and C-flags are affected. The contents of the X-register are not affected by CPX. The comparison is performed by subtracting the contents of the addressed memory location from the contents of the X-register, but the result is not stored in either the X-register or the memory location. The PSR flags are affected as follows:

1. The C-flag will be *set* (C = 1) if the absolute value of the X-register is equal to or greater than the value fetched from memory (X M). The C-flag is *reset* (C = 0) if X is less than the value from memory.
2. If bit 7 of the comparison result is 1, then the N-flag is set (N = 1), but if bit 7 is 0, then the N-flag is reset (N = 0).
3. The Z-flag is set (Z = 0) if the value memory is equal to the value from the X-register, otherwise it is reset (Z = 0).

The CPX instruction can be used for setting the PSR flags, etc.

CPV (Compare Memory with Y-Register). This instruction is identical with the CPX instruction with the exception that the Y-register is used instead of the X-register. Read the discussion on CPX for details that also affect CPV.

DEX (Decrement X-Register). The DEX is a 1-byte instruction that uses implied addressing, i.e. X-register is implied. Execution of DEX will cause the X-register to be reduced, i.e., decremented by 1. Symbolically, this instruction acts as follows:

$$X - 1 \rightarrow X$$

Thus, we can see that the result of DEX is stored in the X-register. The DEX instruction affects only the N and Z-flags of the PSR. If the result of DEX is such that bit 7 is 1, then the N-flag is set (N = 1). If bit 7 of the X-register is 0, then the N-flag is reset (N = 0). The Z-flag is set (Z = 0) if the result of DEX is zero. If the result is non-zero, then the Z-flag will be reset (Z = 0).

DEY (Decrement Y-register). This instruction is exactly like DEX, except that the Y-register is used instead of the X-register. Read the DEX description for details which also apply to DEY.

INX (Increment X-register). The INX instruction is a 1-byte, implied addressing instruction which increases the value stored in the X-register by 1, with the result stored in the X-register. Symbolically:

$$X + 1 \rightarrow X$$

The C and V-flags are not affected by the INX instruction. The N-flag will be set (N = 1) if bit 7 of the X-register is 1, and reset (N = 0) if bit 7 is 0. The Z-flag will be set (Z = 1) if the result of INX is zero, and reset (Z = 0) if the result is non-zero.

INY (Increment Y-register). This instruction is the same as INX, except that the Y-register is used instead of the X-register. See the discussion of INX for details which also apply to INY.

JMP (Jump to Another Memory Location). The JMP instruction causes an immediate transfer of program control to another memory location. Both Absolute and Indirect addressing modes can be used. Symbolically, the JMP instruction is:

$$(PC + 1) \rightarrow PCL$$
$$(PC + 2) \rightarrow PCH$$

Let's consider an example of each form of the JMP instruction. In the Absolute addressing mode, the program counter is loaded with the address given in the following two bytes. Recall that the contents of the program counter determine the location of the next instruction to be executed; for example:

Byte 1	0500H	4CH	JMP nnnn
Byte 2	0501H	52H	nn (low-byte)
Byte 3	0502H	EFH	nn (high-byte)

In this example, a JMP EF52H instruction is encountered at memory location 0500H. Immediately after the execution of JMP EF52H, the contents of the PC will be EF52H and that will be the location of the next instruction to be executed.

An indirect JMP example is:

Byte 1	0500H	6CH	JMP (nnnn)
Byte 2	0501H	B4H	nn
Byte 3	0502H	EFH	nn
	.		
	.		
	.		
	EFB4H	55H	low-order byte
	EFB5H	06H	high-order byte

In this case, a JMP (nnnn) instruction is encountered at 0500H. The parenthesis around the address "nnnn" tells us that the 6502 program counter is loaded not with "nnnn," but rather with the contents of "nnnn" and "nnnn+1." In this case, the "nnnn" described by nnnn is EFB4H. This location in memory contains the low-order byte of the Jump destination address, while EFB5H contains the high-order byte of the destination address. The data bytes in these locations are stuffed into PCL and PCH of the program counter, respectively, to form address 0655H as the destination address.

JSR (Jump to Subroutine). A subroutine is a program or program segment which may be used frequently or only if certain special conditions are met (among other applications). An example is a printer output routine. In a typical scenario, we would load the hexadecimal equivalent code of an ASCII alphanumeric character to be printed into the accumulator, and then jump to the printer subroutine with a JSR instruction. Only Absolute addressing is allowed, and no PSR flags are affected.

The difference between JMP and JSR is that the JSR will store the two bytes of the last instruction address to be executed on the stack, and decrement the Stack Pointer by 2. When the program returns from the subroutine (by encountering an RTS instruction), the program counter will be loaded with the address of the next instruction to follow JSR. Symbolically, on return from subroutine:

$$\text{PCL} + 1 \rightarrow \text{PCL}$$
$$\text{PCH} + 2 \rightarrow \text{PCH}$$

An RTS is always the last instruction in the subroutine, and causes restoration of the main program LDY (Load Y-Register). The LDY instruction is the same as LDX, except that the Y-register is used instead of the X-register. The symbolic notation is:

$$M \rightarrow Y$$

Addressing modes used by LDY are Immediate, Absolute, zero page, zero page X, and Absolute indexed X. Only the N and Z-flags are affected. LDY will set (N = 1) the N-flag if the result makes bit 7 of the Y-register 1, otherwise, N = 0. The Z-flag is set (Z = 1) if the value loaded into the Y-register is zero, otherwise Z = 0.

NOP (No Operation). This 1-byte instruction uses the Implied addressing mode, and does nothing to anything in the 6502 but use up two clock cycles of time and increment the program counter by 1.

PHA (Push Accumulator on Stack). This instruction pushes the contents of the accumulator onto the external stack in memory. The addressing mode is Implied. The Stack Pointer is decremented by 1. No flags are affected, and PHA is a single-byte instruction.

PHP (Push Processor Status on Stack). This instruction is exactly like the PHA instruction, except that the contents of the Processor Status Register are pushed to the stack, instead of the accumulator contents.

PLA (Pull Accumulator from Stack). The PLA instruction is opposite PHA. The Stack Pointer is incremented by 1, and the values stored at that point in the external memory stack are transferred back into the accumulator of the 6502. Addressing mode is Implied. The PLA instruction affects the N and Z-flags of the PSR. The N-flag is set (N = 1) if bit 7 of the accumulator is 1, and reset (N = 0) if bit 7 is 0. The Z-flag will be set (Z = 1) if the value restored to the accumulator is zero, and reset (Z = 0) if the accumulator contents are non-zero.

PLP (Pull Processor Status from Stack). This instruction is similar to PLA, except that the contents of the PSR are restored instead of the accumulator contents. All PSR flags are affected, and will take on the values stored on the stack. PLP increments the Stack Pointer by 1.

RTI (Return from Interrupt). The purpose of this instruction is to restore the 6502 to its previous status, i.e., the status before the interrupt occurred. When the 6502 responds to an interrupt, it pushes the contents of the program counter and Processor Status Register onto the external stack. The RTI instruction pulls these data back from the stack, and restores them to the PC and PSR. Thus, RTI will force the 6502 to restart in the program where the interrupt occurred. The RTI instruction, therefore, should be the last instruction in the interrupt service subroutine program. Otherwise, the 6502 will *not* return to the main program.

RTS (Return from Subroutine). This instruction is analogous to the RTI, except that it is used at the end of a subroutine called by a JSR instruction. The 6502 saves PC and PSR data on the stack in response to the JSR instruction. RTS will restore these data to the 6502 PC and PSR. The RTS must be the final instruction in the subroutine program.

SEC (Set Carry Flag). The SEC instruction causes the Processor Status Register carry flag (C-flag) to be set (C = 1). No other flags or registers are affected. Addressing mode is applied.

SED (Set Decimal Mode). The SED instruction places the 6502 in the decimal mode by setting the decimal flag in the PSR (D = 1). Following this instruction, all SBC and ADC instructions will use BCD arithmetic. No other flags or registers are affected.

SEI (Set Interrupt Disable). The SEI instruction sets the Interrupt Disable flag in the PSR (I = 1). The effect is to prevent the 6502 from responding to interrupt requests on the $\overline{\text{IRQ}}$ line (requests on $\overline{\text{NMI}}$ are not affected). Addressing mode is implied. No other flags or registers are affected.

STY (Store Y-Register in Memory). This instruction stores the contents of the Y-register in a memory location specified by the bytes following the op-code. Allowable addressing modes are Absolute, zero page, and zero page X. No PSR flags are affected.

TAX (Transfer Accumulator to X-Register). This instruction transfers the contents of the accumulator to the X-register. Implied addressing is used. Only the N and Z-flags of the PSR are affected. The N-flag is set (N = 1) if bit 7 of the X-register becomes 1, and reset (N = 0) if bit 7 becomes 0. The Z-flag is set (Z = 1) if the result is zero, and reset (Z = 0) if the result is non-zero. No other flags or registers are affected.

TAY (Transfer Accumulator to Y-Register). This instruction is exactly the same as TAX, except that data destination is the Y-register instead of the X-register.

TYA (Transfer Y-Register to Accumulator). This instruction transfers the contents of the Y-register to the accumulator. Addressing mode is Implied. Only the N and Z-flags of the PSR are affected (see discussion on TAX).

TSX (Transfer Stack Pointer to X-Register). The TSX instruction will transfer the contents of the 6502 Stack Pointer (SP) register to the X-register. Only the N and Z-flags of the PSR are affected (see discussion on TAX). Implied addressing is used.

TXA (Transfer X-Register to the Accumulator). This instruction transfers the contents of the X-register to the accumulator, and is exactly the opposite of TAX. Only the N and Z-flags are affected (see discussion on TAX).

TXS (Transfer X-Register to Stack Pointer). This instruction is opposite the TSX instruction, and will transfer the contents of the X-register to the 6502 Stack Pointer register. Only the N and Z-flags are affected (see discussion on TAX).

We can sometimes use TSX and TXS to relocate the external stack. A typical sequence might be:

```
TSX          Save SP at location
STX,nnnn     nnnn
LDS #aa      Load new SP top location (#aa) in X-reg-
             ister
TXS          Transfer X-register to SP
```

8

65xx-Family Support Chips

The manufacturers of the 65xx (6502) microprocessor chips also offer certain special I/O and other chips which will aid in making an efficient, low-cost computer. In this chapter, we will examine a few of the most common and popular of these chips.

6522

The 6522 Peripheral Interface Adapter (PIA) is 40-pin DIP integrated circuit that contains all the logic to implement I/O functions, with complex handshaking routines, and timer functions. In addition to the standard pair of 8-bit I/O ports, the 6522 also offers a pair of interval timers, a shift register that is useful for serial-to-parallel and parallel-to-serial data conversions.

The 6522 is designed to operate with the 6502 microprocessor, so is often encountered in microcomputers from small single-board OEM models intended to be installed in larger instruments, to full-scale microcomputers with the regular complement of peripheral devices. As a 6502 adjunct, the 6522 is intended for memory-mapped operation. The four address lines on the 6522 are identified in Figure 8-1 as RS0 through RS3. These lines form a 4-bit address that is capable of uniquely addressing up to 16 different internal memory-mapped functions. The 6522 functions are located at the following internal addresses:

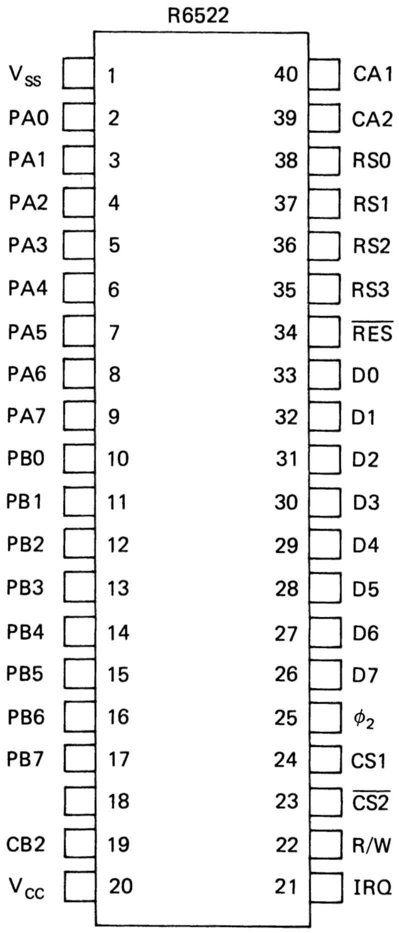


Figure 8-1. 6522 pinouts

Address				*Register*	
RS3	***RS2***	***RS1***	***RS0***	***Designation***	***Comments***
0	0	0	0	ORB	
0	0	0	1	ORA	Controls handshaking
0	0	1	0	DDRB	
0	0	1	1	DDRA	
0	1	0	0	T1L-L, T1C-L	Timer-1 write latch and read counter
0	1	0	1	T1C-H	Trigger T1L-L/T1C-L transfer
0	1	1	0	T1L-L	
0	1	1	1	T1L-H	

Address				*Register*	
RS3	***RS2***	***RS1***	***RS0***	***Designation***	***Comments***
1	0	0	0	T2L-L/T2C-L	Timer-2 write latch and read counter
1	0	0	1	T2C-H	Trigger T2L-L/T2C-L transfer
1	0	1	0	SR	
1	0	1	1	ACR	
1	1	0	0	PCR	
1	1	0	1	IFR	
1	1	1	0	IER	
1	1	1	1	ORA	No effect on handshake

The 6522 is memory-mapped, so will be treated by the microprocessor chip as if it were a bank of 16 bytes of memory. In the AIM-65 microcomputer, for example, the 6522 is memory-mapped at locations A000 through A00F (hex addresses). If we want to write a word to port-A, then we would want to address ORA at location 0001, which in the AIM-65 is A001H.

The configuration of the 6522 ports is interesting and most useful. The port registers are designated ORA (port-A) and ORB (port-B). These Output Registers can be configured as either input or output, on a bit-by-bit basis, under program control. The control mechanism resides in the related Data Direction Registers A and B (DDRA and DDRB). If we want to make all bits of either register an output, then we will write a "1" to the corresponding DDR. Similarly, if we want the register to act as an input, then a "0" is written to the DDR. Thus, to make ORA an output port, we will write FFH to location 0011 (DDRA) of the 6522. If we want the port to be an input port, then we would have written 00H to location 0011H instead of FFH.

The interesting thing about the 6522 output registers is that we may make the ports either inputs or outputs on a bit-for-bit basis. Thus, we can make B0 an input, B1 an output, etc. All we need do is write the correct word to the selected DDR that will configure the individual bits as needed. Suppose, for example, we wanted to configure the bits of ORB as follows:

ORB Bit	***Function***	***DDRB State***
PB7	Input	0
PB6	Input	0
PB5	Output	1
PB4	Input	0
PB3	Output	1
PB2	Output	1

ORB Bit	*Function*	*DDRB State*
PB1	Output	1
PB0	Input	0

Thus, if we write the binary word 00101110_2 (i.e., 2EH) to DDRB at location 0010H of the 6522, ORB will be configured as shown. We can also configure ORA as needed using a similar scheme modified to meet the needs of the user. This is done under program control. If the function of each bit of both ports remains immutable, then the programming chores can be accomplished once when the computer is first turned on, or reset. The initial program steps will be housekeeping in nature, and may well include setting up ORA and ORB by programming DDRA and DDRB.

The 6522 pinouts are discussed here:

Designation	*Pin*	*Description*
ϕ_2	25	Phase-2 clock input. This clock regulates the transfer of data between the PIA and the system (transfer on ϕ_2 = HIGH), and serves as the timer base for on-chip timers and shift registers (SR).
CS1, $\overline{\text{CS2}}$	24, 23	Chip-select lines. CS1 is active-HIGH, $\overline{\text{CS2}}$ is active-LOW. Both lines must be active for chip to be on.
RS0-RS3	38, 37, 36, 35	Register-select lines. These lines address the internal functions of the 6522, and are normally connected to bits of the address bus as dictated by system memory map.
R/$\overline{\text{W}}$	22	Read/write line. A HIGH indicates that data are being transferred out of the 6522 to the system; a LOW indicates data will be transferred into the system. This line is a control input, and will not affect the 6522 unless CS1 is HIGH and $\overline{\text{CS2}}$ is LOW.
D0-D7	33-26	Data bus lines. Data will be transferred to and from the 6522 over these lines if the chip-select, R/$\overline{\text{W}}$ and ϕ_2 = HIGH criteria are met.
$\overline{\text{RES}}$	34	Reset. Active-LOW input that will clear (i.e., set = 0) all registers except T1, T2, and SR.
$\overline{\text{IRQ}}$	21	Interrupt request. This active-LOW output will go LOW when both the interrupt enable bit and interrupt flags of the 6522 are set (= 1). This pin is used for such purposes as signalling

Designation	Pin	Description
		the processor that a timer interval has expired.
PA0-PA7	2-9	Peripheral interface for port-A. The input and/or output pins for port-A.
PB0-PB7	10-17	Peripheral interface for port-B. The input and/or output pins for port-B.
CA1, CA2	40, 39	Peripheral control lines for port-A. These lines act as either interrupt lines or as handshaking lines. Operation is controlled through the Internal Control Register (ICR).
CB1, CB2	18, 19	Peripheral control lines for port-B (see above CA1, CA2). In addition, these lines act as the serial port for the shift register (SR).

6530

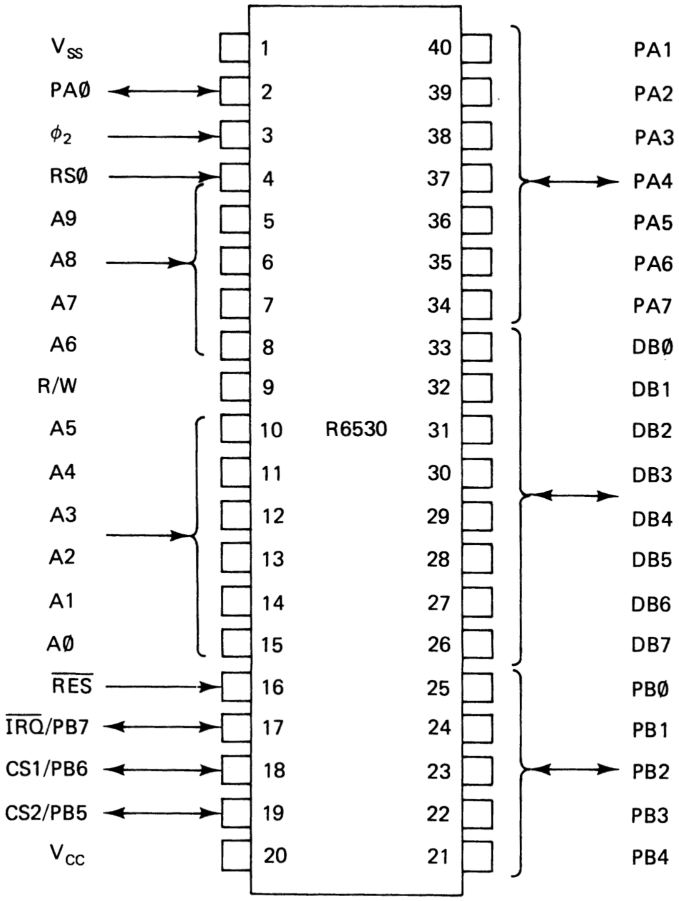
The 6530 device is a ROM-RAM-I/O timer integrated circuit. The device contains a mask-programmable *read only memory* (ROM) that will store up to 1024 8-bit words. It also contains a 64 byte 8-bit *random access memory* (RAM), two 8-bit bidirectional I/O ports, and a programmable interval timer. The 6530 device is, therefore, extremely versatile. The interval timer will time various intervals from 1 to 262,144 clock periods, and is under software control in the I/O configuration. The 6530 device contains an 8-bit bidirectional data bus for communication between the "outside world" and the 8-bit data bus of the microprocessor. There is also a pair of 8-bit buses for communication with external peripheral devices. All lines are both TTL- and CMOS-compatible.

The 6530 architecture is divided in four main sections within the IC: RAM, ROM, I/O, and timer. The I/O section consists of the two 8-bit portions discussed here, and are controlled by a pair of *data direction registers* (DDR), designated "A" and "B." This form of architecture permits us to configure both ports as either input or output on a bit-for-bit basis. Thus, a HIGH (logical-1) written to a bit of the DDR will cause the corresponding bit of the associated port to be configured as an output. Similarly, writing a LOW to the DDR bit makes the corresponding port bit an input.

The 6530 device contains two forms of internal memory. There is a 64-byte by 8-bit RAM, which will permit both read and write operations. This memory can be used as a "scratchpad" memory. There is also a 1K by 8-bit ROM which is addressed by A0-A9 and RS0 (see Figure 8-2). By using CS1 and CS2, we can parallel up to seven different 6530 devices.

Pinouts for the 6530

Designation	*Pin*	*Description*
$\overline{\text{RES}}$	16	Reset. This active-LOW line will cause all I/O registers to clear, causing all I/O lines to act as inputs. The $\overline{\text{RES}}$ line must remain LOW for not less than one complete clock cycle.



R6530 Pinout Designation

Figure 8-2. 6530 pinouts

Pinouts for the 6530 (continued)

Designation	*Pin*	*Description*
Φ_2	3	Phase-two clock. This line connects to the phase-2 clock of the 6502 microcomputer system. The LOW state of the clock will be any potential from 0 volts to 0.4 volt, while a HIGH will be +2.4 or more volts.
R/$\overline{\text{W}}$	9	Read/Write. When this line is HIGH, the 6502 will be able to read data from the 6530, while a LOW allows the 6502 to write data to the 6530.
$\overline{\text{IRQ}}$	17	Interrupt Request. Also used as PB7 in non-timer modes.
DB0-DB7	33-26	Data bus
PA0-PA7	2, 40-34	Peripheral data bus "A"
PB0-PB7		Peripheral data bus "B"

9 Device Selection and Address Decoding

The control and timing signals synchronize the operation of the 6502-based microcomputer. Such an arrangement is necessary when numerous (up to 65,536) devices can share a common 8-bit data bus. The information provided by the control signals concerns what device is being called upon and what it is supposed to do. The control and timing signals arbitrate the use of the bus in response to the instructions provided by the programmer. In this section, we will discuss how these signals are used to designate and instruct the memory and peripheral devices connected to the 6502 bus.

Two jobs must be done by the 6502 control signals: First, it must designate the device that is selected, and second, tell it whether a read or a write is to take place. The address bus designates not only memory locations but also I/O ports (the 6502 uses memory-mapped I/O). Since the address bus contains 16 parallel bits, the bus can uniquely address 2^{16}, or $65{,}536_{10}$, different memory locations or peripherals. Valid memory addresses range from 00000000_2 (00H) to 11111111_2 (FFH). The designation of read or write is indicated by the status of the R/W line on the 6502 during phase 2. Thus, we can select any device, whether memory or memory-mapped peripherals by using the address bus, the R/W line, and the phase-2 clock signals.

ADDRESS DECODING

The purpose of an address decoder is to provide a signal that becomes active only when the correct address is on the address bus. Decoders may use all 16 bits (A0-A15) of the bus, or just 1 or 2 bits. In one scheme only a single bit is needed to turn on a teletypewriter. In that case, the computer only had 26K of memory so the A15 bit never came on to address active memory. The A15 bit defines the 32K boundary (80H = 32K), so will come on only when addressing a location of 32K or higher. Thus, since there is no memory or other peripherals above 26K, we can use A15 to joggle the teletypewriter/printer on and off. An example will be given in Chapter 11. Most address decoders require more than a single bit.

An address decoder requires some means of examining multiple-bit lines and deciding which of two possible outputs to issue. The address decoder may have an active-HIGH output (goes HIGH when the correct address is present) or an active-LOW output (LOW on correct address). The 7530 TTL chip is a popular device in decoder circuits (see Figure 9-1).

The 7430 is an 8-input NAND gate. The rules which govern the operation of any NAND gate are:

1. If any one input is LOW, then the output is HIGH.
2. All 8 inputs must be HIGH for the output to be LOW.

Thus, in order to use the 7430 as an 8-bit address decoder, we must somehow conspire to make all 8 bits HIGH (logical-1) when the correct address is on the bus. The only combination where that situation arises naturally is 11111111_2 (FFH). For all other addresses we must provide inverters on each address line where zeros are expected. Figure 9-2 shows a sample 7430 decoder circuit for address 11001011_2 (CBH). Since bits A0, A1, A3, A6, and A7 are already 1 when this address is presented, nothing else need be done—the address lines can be connected directly to the inputs of the 7430. For bits A2, A4, and

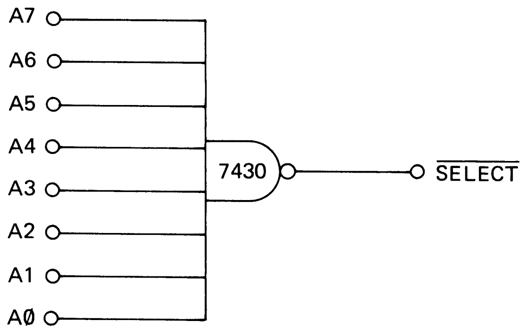


Figure 9-1. Eight-bit address decoder based on the 7430 chip

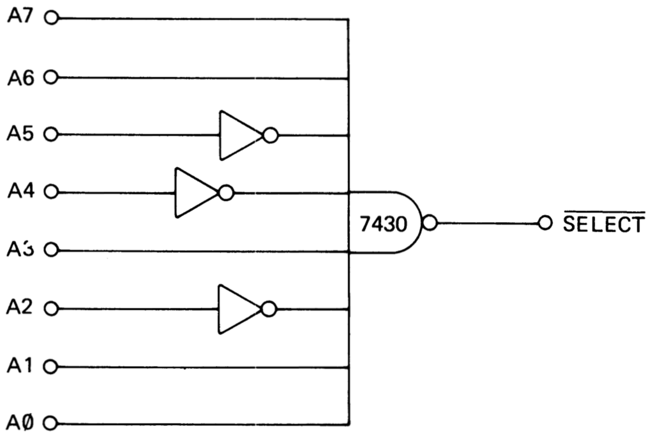


Figure 9-2. Practical version of Figure 9-1

A5, however, a different tactic is required. These bits will be LOW when the correct address is presented, so must be inverted. Thus, an inverter is provided in each of these address bus lines so that the 7430 will see 11111111_2 when 11001011 is present on the address bus.

The output of the 7430 is an active-LOW signal which we designate SELECT. When the correct 8-bit address appears on the bus, this signal drops LOW; at all other times it is HIGH.

The circuit in Figure 9-2 is capable of seeing only 8 bits of the 16-bit address bus. If we want to examine all 16 bits, then some other tactic is needed; an example is shown in Figure 9-3. In this circuit, two 7430 devices are used in combination. One 7430 examines bits A0–A7 while the other examines A8–A15. The outputs are combined in a 2-input NOR gate. The rules for a NOR gate are:

1. If either input is HIGH, then the output is LOW.
2. Both inputs must be LOW for the output to be HIGH.

Since both 7430s have active-LOW outputs, output of the 7402 will be HIGH only when the correct address is present on the inputs of the 7430 devices. All other but the correct inputs will cause one or the other 7430 output (or both!) to be HIGH, thereby forcing the 7402 output LOW. The circuit in Figure 9-3 would not be used very often for several reasons. Among them is the fact that we could sometimes use schemes which reduce the number of address bus lines needed to uniquely designate a memory or I/O port location.

Another type of address decoder is shown in Figure 9-4. This device is based on the TTL 7442 device which is known as a BCD-to-

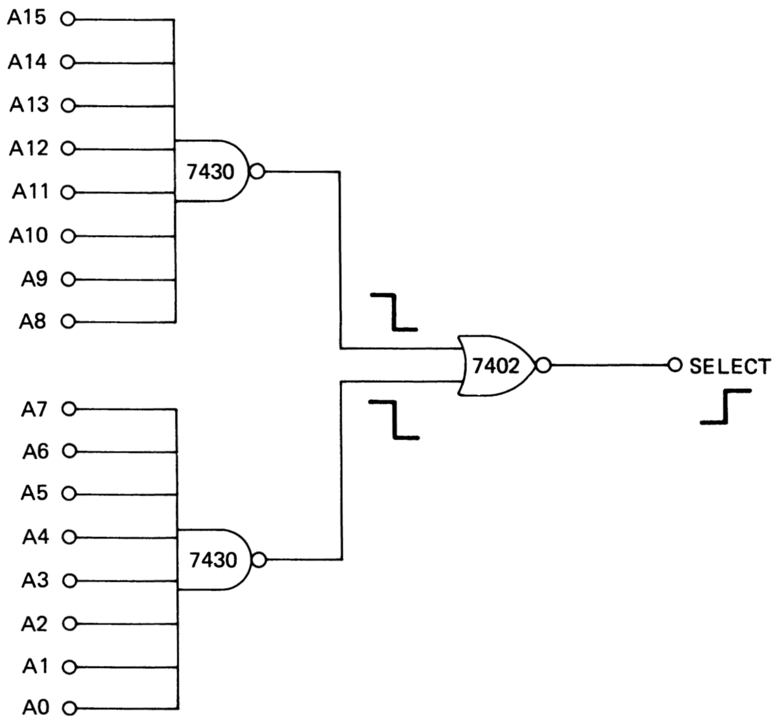


Figure 9-3. Eight-bit decoders expanded to sixteen-bit operation

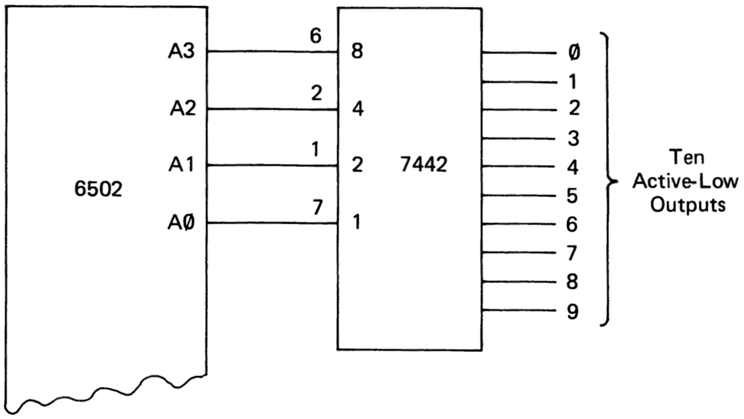


Figure 9-4. 7442 used as an address decoder

1-of-10 decoder. The 7442 was originally designed to provide decimal (1-of-10) output in response to Binary Coded Decimal (BCD) input. The BCD code uses a 4-bit binary number to represent decimal digits. The normal weighting for the 4 bits is the 8-4-2-1 weighting of any 4-bit binary "word." The BCD codes and their decimal equivalents are:

BCD	*Decimal*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

There are ten unique output lines on the 7442 decimal, one for each decimal digit. When a 4-bit BCD word is applied to the inputs, the corresponding decimal output will drop LOW. Thus, when the binary (BCD) code 0011_2 is applied to the inputs, output 3 drops LOW; all other outputs remain HIGH.

Figure 9-4 shows a 7442 device connected to the low-order 4 bits of the address bus. The following devices are selected:

A3	*A2*	*A1*	*A0*	*Device*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9

We can, therefore, select up to ten devices using a single 7442. They do not have to be on the A0–A3; we can use any four bits of the address bus if we want to locate the device elsewhere. We could, for example, connect the "8" input of the 7442 to the A15 bit, and then

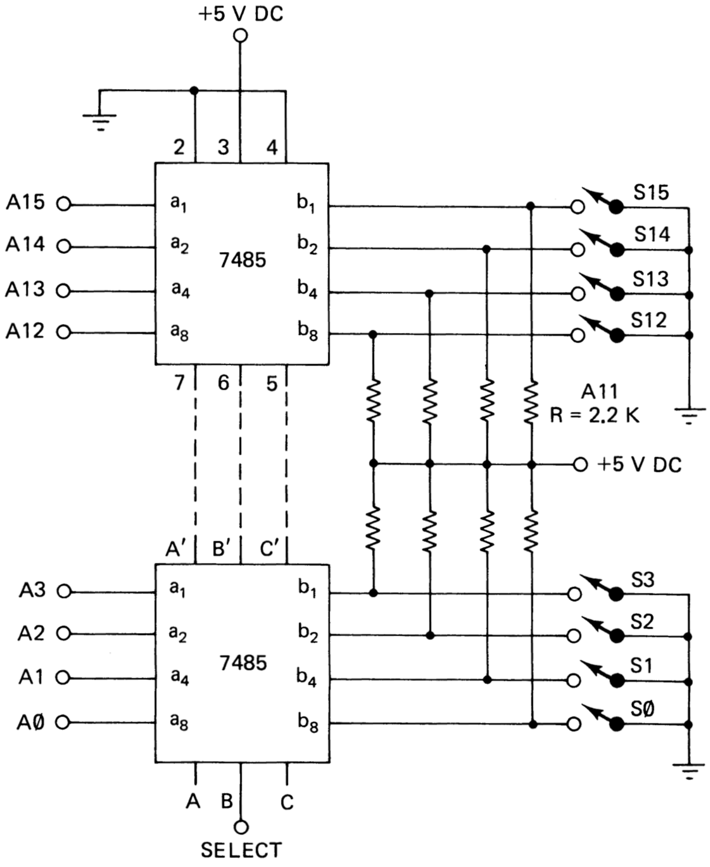


Figure 9-5. 7485 chips used in 4-bit banks for address decoding

connect A0, A1, and A2 to the 1-2-4 inputs of the 7442. In this way we could locate ten I/O ports starting at the 32K boundary.

Two or more 7442 devices can be used together to provide the ability to look at 4, 8, 12, or 16 bits. We could combine the selected outputs in a 2, 3, or 4-input NOR gate, as in Figure 9-3. We could also use a cascading arrangement, which will be discussed in Chapter 10 under the heading of "banking."

Another device which is sometimes used as an address decoder is the 7485 four-bit magnitude comparator, or its CMOS pin-for-pin equivalent, the 4063. These devices examine two 4-bit binary words, designated A and B, and issue unique outputs that indicate whether A equals B, A is less than B, or A is greater than B. If we program one set of inputs (e.g., B) with the desired address code, then we can use the A = B output as a SELECT signal. In that case, the A inputs are connected to the address bus lines. Figure 9-5 shows the circuit which can be used for any bit length (16 bits are shown).

The 7485 is equipped with cascade inputs that are used to join two or more devices together to form longer words. If an increment of 4 bits is desired, then all inputs of the 7485 are used. We can use less than 4 bits by strapping the same unused inputs on both A and B words to the same level. It doesn't matter whether you strap them HIGH or LOW, so long as corresponding inputs on both sides of the same chip are at the same level. Each additional 7485 will extend the address word length from 1 to 4 bits.

In some cases the programmed inputs will be permanently wired HIGH or LOW according to the bit pattern required by the designated address. In other cases, we will want to vary the address occasionally, so will use switches as in Figure 9-5. Each input is equipped with a pull-up resistor to V+ and a switch; when the switch is open, the input is HIGH; when the switch is closed, the input is LOW (grounded).

Rarely do we need all 16 bits to designate an address. Figure 9-6 shows a method using an 8-bit decoder (any of the circuits can be used, not just the 7430) combined with a 2-bit (7400) decoder to perform a specific chore, e.g., I/O decoder. The address that this circuit responds to will be above $49{,}152_{10}$ because the A14 and A15 bits must be on. In addition, the SELECTL signal for 00110111_2 must also be

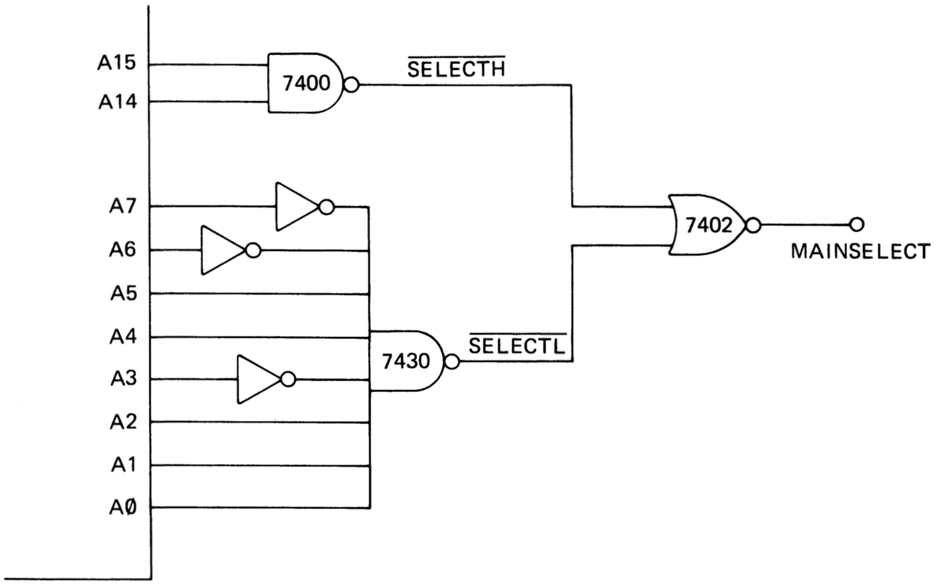


Figure 9-6. Simplified address decoding when not all memory is used

true (LOW). If both SELECTL and SELECTH are LOW, then the output of the 7402 NOR gate will be HIGH (a signal designated as "MAIN-SELECT!").

GENERATING READ/WRITE SIGNALS

The 6502 indicates read and write conditions by the coincidence of the phase-2 signal with the state of the R/W line. For a read operation, both the phase-2 line and the R/W are HIGH. For a write operation, the phase-2 clock is HIGH and R/W is LOW. In order to generate unique and discrete READ and WRITE signals, we must take into account both phase-2 and R/W lines. These new signals must also be capable of driving enough TTL roads for the planned size of the computer. Given the nature of some machines, the total fan-out might be 100 or more. For most applications, however, the standard fan-out of 10 offered by most TTL devices is sufficient. Where higher drive capability is needed, we can use bus driver ICs, which have high fan-outs.

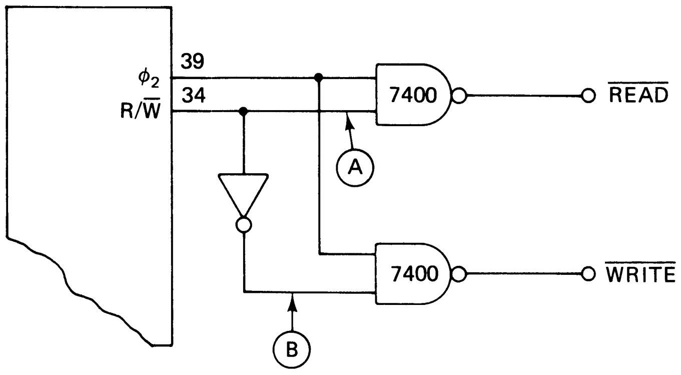
Figure 9-7A shows a simple READ and WRITE signal generator circuit that has a fan-out of 10. Both outputs are produced by TTL 7400 NAND gate sections (the 7400 contains four independent 2-input NAND gates). The usual rules apply:

1. A LOW on any one input will cause the NAND output to be HIGH.
2. Both inputs must be HIGH for the output to be LOW.

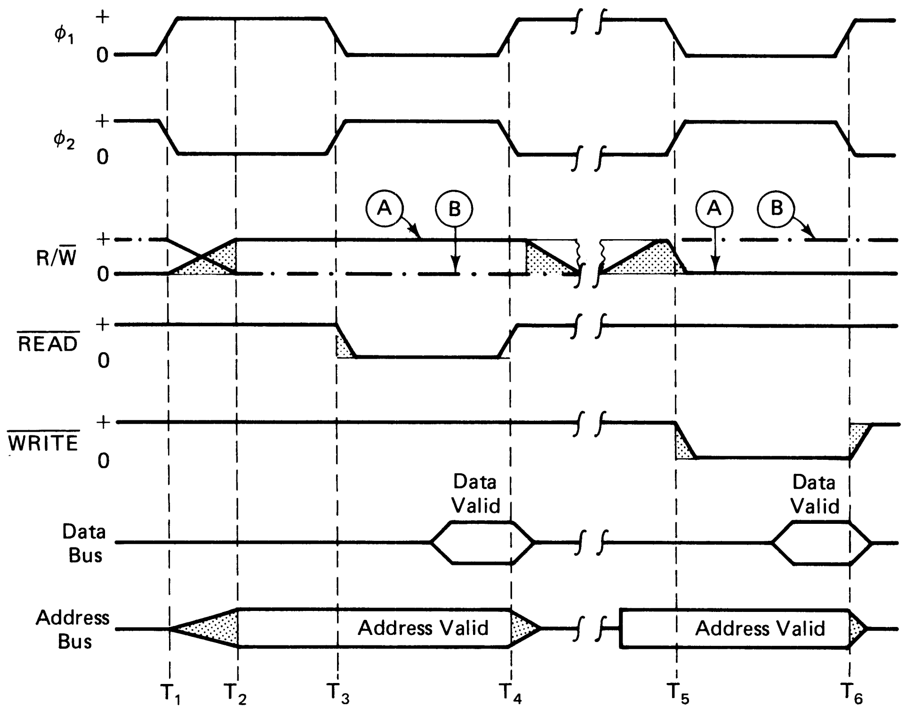
Since both phase-2 and R/W outputs are HIGH to read operations, we can generate our system READ by connecting these lines to the inputs of the NAND gate. The output of that NAND gate will drop LOW, forming a READ signal, but only when phase-2 and the R/W line are both HIGH.

The system WRITE signal is also generated by the phase-2 and R/W lines, but requires the R/W line be inverted first. Figure 9-7B shows the timing diagram for both read and write operations. Note that the R/W line is shown in both normal (A) and inverted (B) forms. The times for this diagram are the same as those in Chapter 4.

The phase-1 clock cycle starts at time T_1. The address of the selected memory location is output on the address bus (A0 through A15) during this period, and becomes stable in about 300 nanoseconds (T_2). The address remains valid until the end of the phase-2 clock cycle. At time T_3, the phase-2 cycle begins, and finds the R/W line HIGH (see A). At this time the READ output drops LOW, and remains LOW



(A)



(B)

Figure 9-7. Generating system $\overline{READ}$ and $\overline{WRITE}$ signals (A) circuits, (B) timing diagram

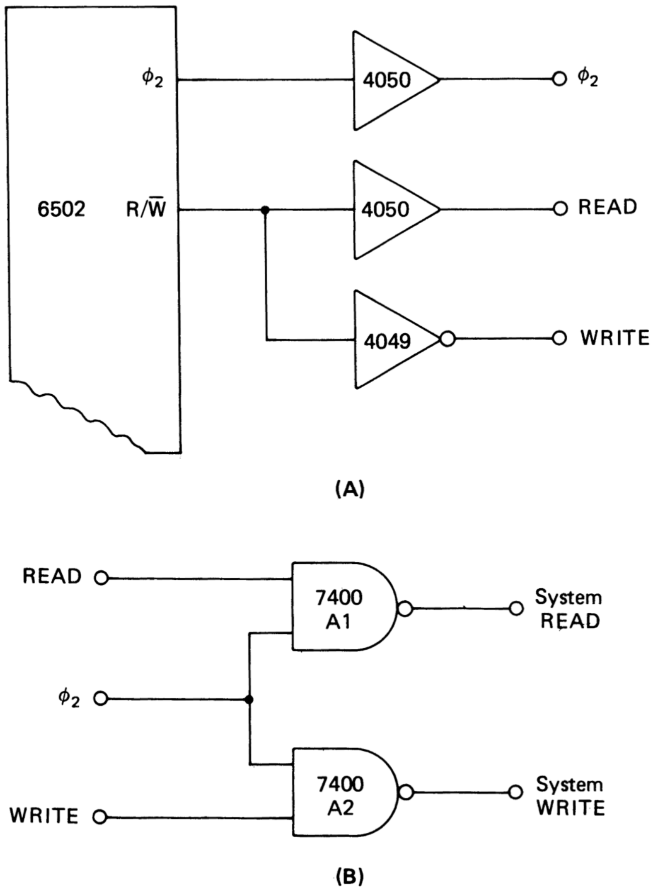


Figure 9-8. A) Buffered phase-2 clock, READ and WRITE signals, B) decoding for system READ and WRITE signals

throughout the phase-2 cycle. At the end of phase-2, the READ line returns to the inactive HIGH state.

The write operation follows a similar routine, but the inverted R/W line (B) must be HIGH for the WRITE signal to be active. At time T_5, the WRITE line goes LOW.

Figure 9-8A shows another method for generating discrete READ and WRITE signals. This method, or one closely related to it, is used extensively in 6502-based microcomputers. The READ signal is buffered by a noninverting buffer device such as the 4050 CMOS device. The WRITE signal is formed, also from the R/W output of the 6502, by an inverter (the example shown is a CMOS 4949). Like the R/W line, the phase-2 line is also buffered by a 4050 CMOS device.

The use of the signal generated by the circuit in Figure 9-8A in creating system READ and system WRITE is shown in Figure 9-8B. A pair of 7400 (or equivalent) NAND gates is used in this circuit. One input from each gate is connected together at the phase-2 line. Thus, when the phase-2 clock is HIGH, the gates are enabled. If the READ signal is HIGH, then the system-READ signal will go LOW. Similarly, if the WRITE signal is HIGH, then the system-WRITE will go LOW.

A slightly different version of this circuit is made by inserting an inverter in the WRITE input line to A2, and then joining the inverter input to the READ input at R/W (see Figure 9-7A).

Figure 9-9 shows the use of a 7442 to create device-select signals in 6502 systems. Recall from our earlier discussion that the 7442 is a BCD-to-1-of-10 decoder. If we apply the control signals to the BCD inputs of the 7442, then we can generate device-select signals. In the scheme shown, the phase-2 signal is applied to the A input (weight = 1), R/W to B (weight = 2), and the SELECT is applied to the C input (weight = 4). We can either ground the D input (weight = 8) or use it as an active-LOW chip select (CS) signal. The coding that the 7442 responds to is:

D	*C*	*B*	*A*	*Active Output*	*Signal*
0	0	0	0	0	(none)
0	0	0	1	1	(none)
0	0	1	0	2	(none)
0	0	1	1	3	(none)

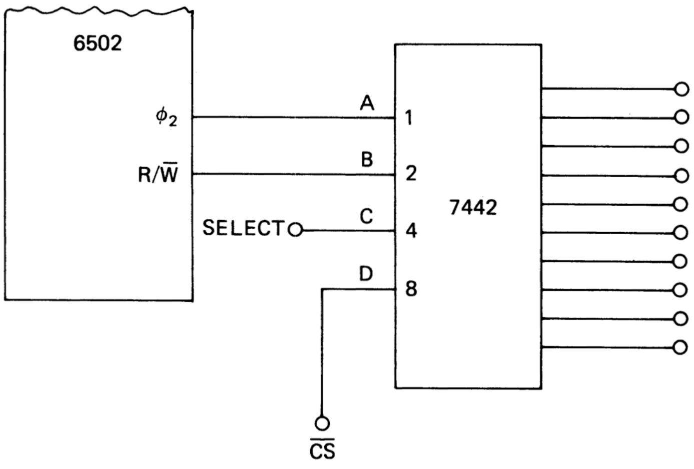


Figure 9-9. 7442 used for system $\overline{READ}$ and $\overline{WRITE}$ signal generation

D	*C*	*B*	*A*	*Active Output*	*Signal*
0	1	0	0	4	(none)
0	1	0	1	5	WRITE
0	1	1	0	6	(none)
0	1	1	1	7	READ
1	0	0	0	8	Chip not selected
1	0	0	1	9	Chip not selected

In the case where an active-LOW SELECT signal is used, the C input will be LOW when the address is selected. In those cases a READ will look like 0011 input to the 7442, so the 3 output would be the READ signal. The WRITE signal would create the code 0001, so the 1 output would be used.

10 Interfacing Memory to the 6502

The very nature of the programmable digital computer, no matter how large or how small, requires *memory*. No sequential or serial processing machine could work unless there were some way to store—or remember—data and programming instructions. Hence, memory devices are inherent in digital computer design.

The principal difference between large mainframe computers and even the smallest single-board computer is essentially one of scale. In terms of memory, this difference translates into certain restrictions on the microcomputer regarding the types of memory devices that are used. The generally slower 8-bit microcomputer, for example, has little need for 20-nanosecond ECL memory elements because the CPU will never operate fast enough to make either efficient or cost-effective use of such memory.

Similarly, the microcomputer probably has no need for multiple disk packs such as found in almost all large mainframe computer installations. For most microcomputers, the simple floppy disk (diskette) is sufficient. As the lines blur between classes of computer, however, the situation may radically change. There are already multi-megabyte large single-disk drives on the market made especially for microcomputers. Several manufacturers offer microcomputers in upright 19-inch racks that look for all the world like minicomputers of not long ago, and these are equipped with "hard disk" drives. One wonders whether the traditional definitions that distinguish minicomputers from microcomputers are still valid. This industry moves too fast for "tradition" to have much meaning; reality keeps changing.

MEMORY HIERARCHY

Various types of memory are still available, and they differ markedly as to the time required to read or write data. We can classify memory into several very broad categories according to approximate access time: cache memory, short-term or "working store" memory, medium-term memory, and long-term memory.

A cache memory is one that operates at ultrahigh speeds, and is used where the memory must keep up with a high-speed central processor. Typical technologies used to form semiconductor cache memories are all high frequency devices: emitter-coupled logic (ECL), high-speed TTL, and current injection logic (IIL or I^2L). As with any circuit that operates in ultrashort periods of time (i.e., 5 to 100 nanoseconds), cache memory designers must be aware of such matters as VHF/UHF circuit layout practices, matching of input and output impedances, and the transmission-line properties of electrical conductors.

Cache memories are usually limited to a small portion of a mainframe computer's total memory array. Data is transferred in and out of the small "cache" as needed.

Short-term memory is the main volatile memory of a microcomputer and consists mostly of semiconductor random access memory (RAM) chips. Short-term memory devices usually operate with access times on the order of 100 nanoseconds to 5 microseconds.

The working store of most microcomputers consists of an array of high-speed short-term devices comprising as few as 32 bytes and as much as hundreds of kilobytes.

The "typical" (if that word can have meaning in this context) 8-bit microcomputer has a 16-bit address bus, so can access 2^{16} (65,536) different 1-byte (i.e., 8-bit) memory locations. Of course, 16-bit machines will have 2-byte circuits at each memory location.

TYPES OF MEMORY DEVICES

Solid-state computer memory devices can be divided into RAM and ROM. RAM is random access (read/write) memory, while ROM is read only memory. The latter type is programmed once, and then installed into the computer as a permanent program or data, while the RAM can be used to either read from or write to; RAM can contain program instructions, data, look-up table entries, etc. RAM devices can be further broken down into static RAM and dynamic RAM.

Figure 10-1 shows both static and dynamic RAM devices in model form. The static RAM is shown in Figure 10-1A. Such a memory device will consist of a flip-flop that can be set to either 1 (HIGH) or 0 (LOW). In this case, a Type-D flip-flop is used, and such a flip-flop will obey the following rules:

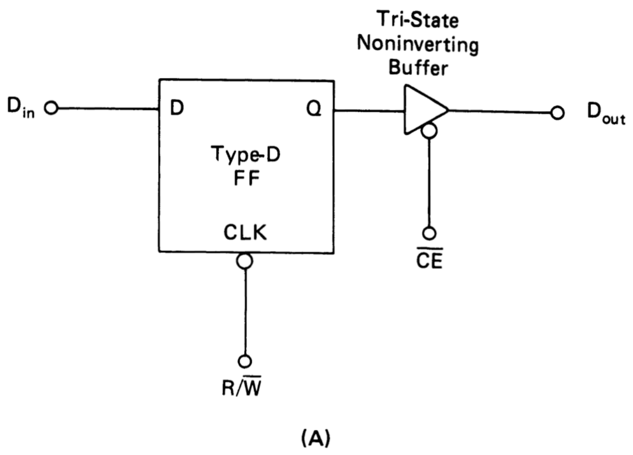
1. When the clock (CLK) line is active (LOW, in this case), the data on the D-input will be transferred to the Q-output.
2. When the CLK input is HIGH (inactive), the data on the Q-output remains at the level it took the last time the CLK line was active. In other words, if a HIGH was present on D-input when the CLK line underwent the transition from LOW to HIGH, then the Q-output will remain at HIGH. Thus, the Type-D flip-flop will "remember" the HIGH condition (convenient? After all, that's what a *memory* element is supposed to do!).

In the case of Figure 10-1A, the D-input of the flip-flop becomes the D_{in} line of the memory cell, and the CLK line becomes a $R/\overline{W}$ line. The Q-output becomes the data output (D_{out}) but only after being passed through a tri-state noninverting buffer stage. This stage is used to keep the Q-output from loading the data bus of the computer unless the computer directs it to be active by issuing the active-LOW chip enable ($\overline{CE}$) signal. There will also be a similar gate at the data input to keep the flip-flop from operating every time the $R/\overline{W}$ line on the 6502 is LOW. In other cases, the $R/\overline{W}$ line on the memory element is connected to a device select circuit rather than the system $R/\overline{W}$ signal.

The static memory device offers the advantage that it will remember the bit of data input to it until it is either rewritten or power is lost on the computer system. But the static memory also suffers from requiring relatively large amounts of electrical current, which can increase considerably the current requirements of the machine. The dynamic memory generally requires less current, and is described here.

Figure 10-1B shows the basic 1-bit memory cell inside a dynamic memory chip. We can model the dynamic memory as a switch-controlled leaky capacitor, in which Q1 (of Figure 10-1B) is the switch. The addressing of the memory cell is made a little easier by arranging them in a row-column matrix in which any one cell is uniquely accessed via a specific BIT line and WORD line. We will discuss addressing more in a moment.

The leakage factor means that the dynamic memory will not hold data indefinitely, but must be refreshed every so many milliseconds. In some cases, the CPU will have to handle that chore, while in others



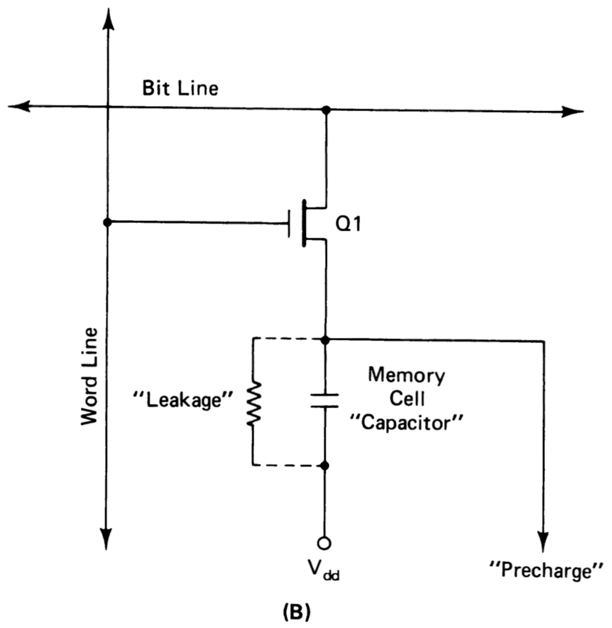


Figure 10-1. A) Type-D flip-flop used as a single-bit memory cell in static RAM, B) dynamic RAM memory cell

on-board refresh capability will be provided that does not require the attention of the 6502.

Figure 10-2A shows an example of a static memory device called by Intel the 8102A, and others the 2102A device. This is 1024 × 1-bit chip, so a bank of eight 2102As will make a 1K 8-bit computer memory. There are ten address lines (A0–A9), as needed, to address 1024 different cells ($2^{10} = 1024$). There are also data input (D_{in}) and data output (D_{out}) lines on this chip, as well as the $\overline{CE}$ and $R/\overline{W}$ lines. Figure 10-2B shows the truth table for the operation of these pins.

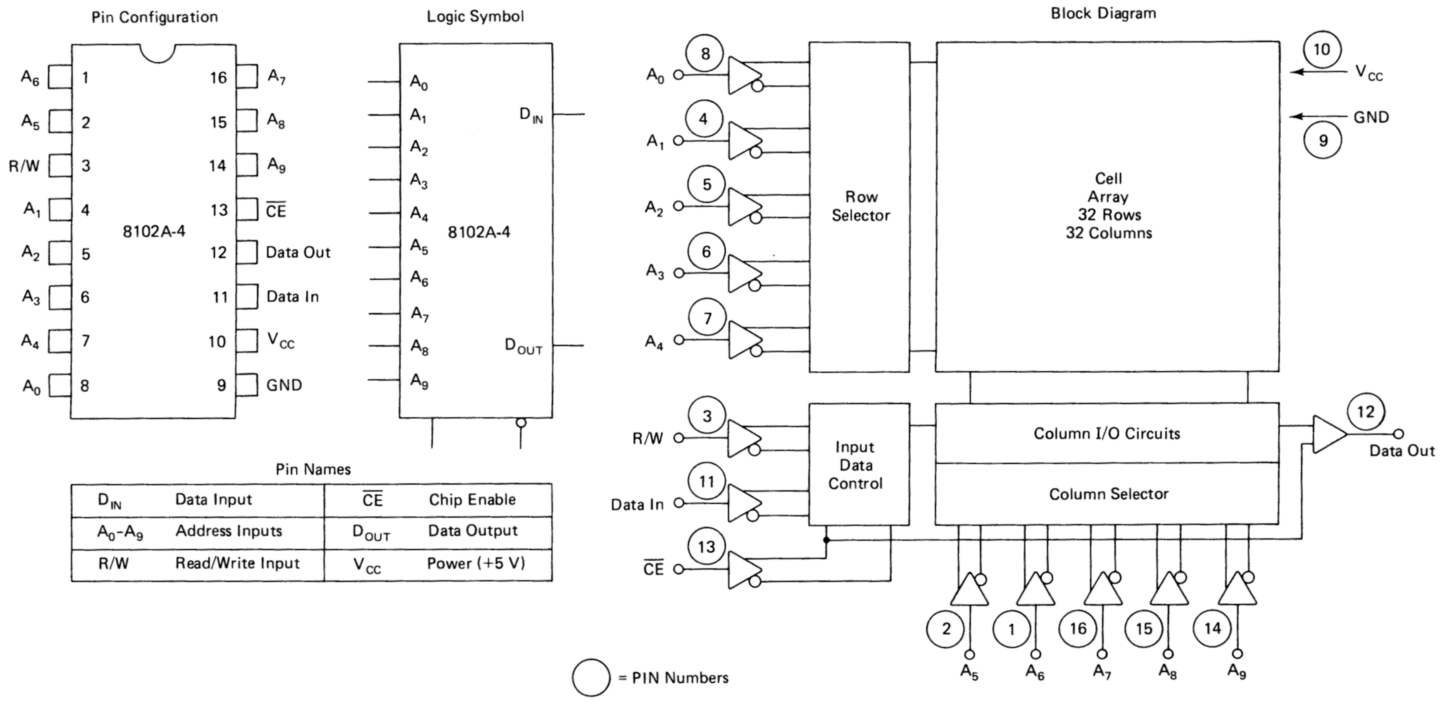
The read cycle (see Figure 10-2C) outputs data from pin D_{out} of the 2102A to the system data bus. There is a certain access time (T_A) required to read data. The read cycle must be at least this long, or data will be lost. For 2102A devices, the nominal T_A is 450 nanoseconds, with selected devices available with 250 nanosecond capability. The 450 nanosecond devices cannot be operated with microprocessor chips whose read cycle is less than 450 nS duration—a very real possibility given the clock speeds of some modern CPU chips. For those cases, the faster chips are mandatory.

The read cycle requires a HIGH on the $R/\overline{W}$ line, and a LOW on the $\overline{CE}$ line. In a real computer, it is likely that the $\overline{CE}$ line will be connected to some sort of bank selector circuit and the $R/\overline{W}$ line to the device select line (see Chapter 9).

The write cycle permits the 6502 to input data into the memory device. In this case, we also require a LOW on the $\overline{CE}$ line, but the $R/\overline{W}$ line must be LOW.

4116 16K × 1-Bit Dynamic RAM

Dynamic RAM provides certain advantages over static RAM, especially in systems with large RAM arrays. For small systems, i.e., those of only a few thousand bytes of memory, static RAM is probably most economical. The DRAM device usually has a density of at least four to one over the static versions, so can be configured in a large array that occupies little space. This factor makes it possible to make small desktop computers that don't generate too much heat to raise the room temperature. The reliability of the computer is also improved when DRAM devices replace static memory in large arrays. The lower heat generation has a lot to do with the reduction in failures, as does the lower parts count, since there are fewer components to fail. As a result of these factors, the use of DRAM devices permits a lower cost, more reliable unit when large arrays are used. The extra cost of the external refresh circuitry does not increase proportionally with memory size,



Pin Names

D_{IN}	Data Input	$\overline{CE}$	Chip Enable
A_0-A_9	Address Inputs	D_{OUT}	Data Output
R/W	Read/Write Input	V_{CC}	Power (+5 V)

(A)

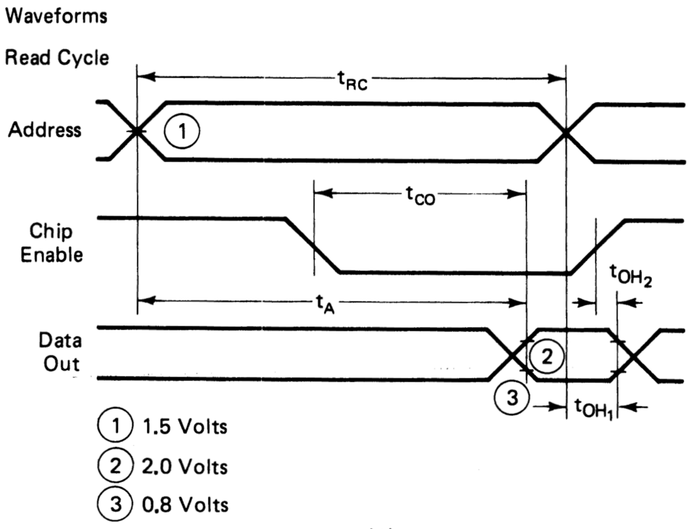
Figure 10-2. A) 1024 × 1-bit static memory chip (copyright 1977 by Intel, used by permission)

$\overline{CE}$	$R/\overline{W}$	D_{IN}	D_{OUT}	Mode
H	X	X	Hi-Z	(Device not Selected)
L	L	L	L	WRITE Ø
L	L	H	H	WRITE 1
L	H	X	D_{OUT}	READ

2102A Truth Table

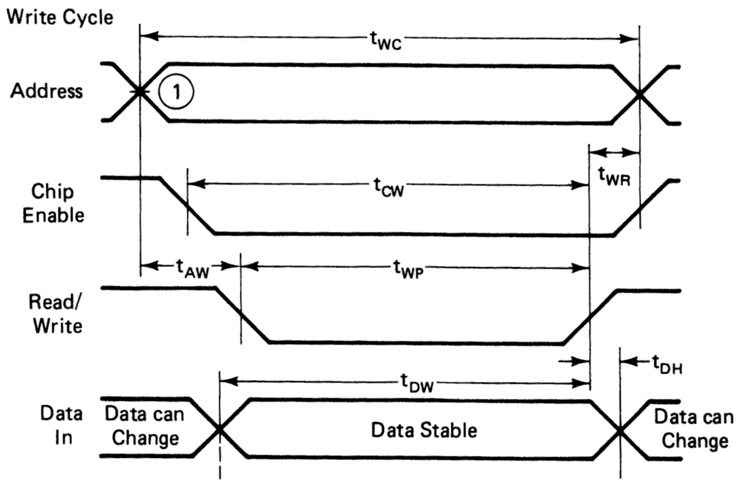
H = High
L = Low
X = Don't Care

(B)



(C)

Figure 10-2 (continued). B) 2102/8102 truth table, C) READ cycle timing



(D)

Figure 10-2 (continued). D) WRITE cycle timing

so is distributed over the entire 64K—making the DRAM more economical in higher order arrays.

Although we are going to discuss one of the most popular DRAM devices, be aware that many offer greater than 16K size. The Fairchild 4164 device is a 64K single-chip DRAM, as is the MCM6664A by Motorola Semiconductor, Inc.

The 4116 device is shown in Figure 10-3. The block diagram of the internal circuitry is shown in Figure 10-3A, while the logic symbol used in schematics is shown in Figure 10-3B and the pinouts/pin names are in Figure 10-3C. Note that the 4116 device only has 7 address bits (A0 through A6). The 16K memory contained within the chip, however, would normally require 14 bits on the address bus. The 4116 overcomes this problem by using a multiplexed addressing scheme in which a row address select ($\overline{\text{RAS}}$) and a column address select ($\overline{\text{CAS}}$) alternately select half of the total address bits required. When the $\overline{\text{RAS}}$ line is LOW, the 7 address lines input the lower order 7 bits into a special 7-bit latch that holds the data. Similarly, when the $\overline{\text{CAS}}$ is made LOW, the high order 7 bits are input to another 7-bit register. Of course, the microcomputer must be designed to connect bits A0 through A6 of the system address bus to the A0–A6 lines of the 4116 on one cycle, and A7 through A13 of the address bus to A0–A6 of the 4116 on the next cycle.

The organization of the 4116 device is an X-Y matrix in which a storage array of 128 horizontal rows contains 128 memory cells each.

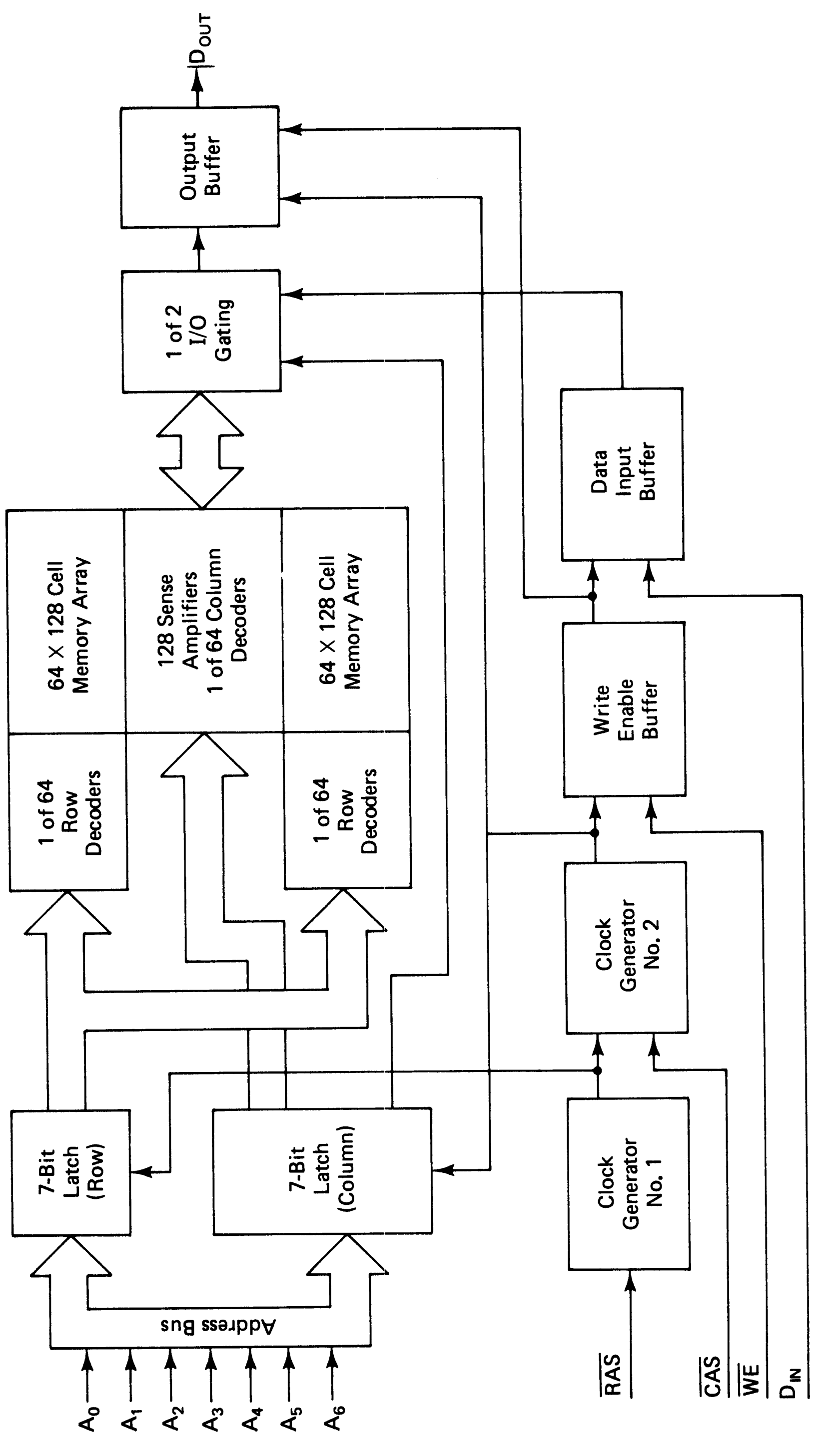
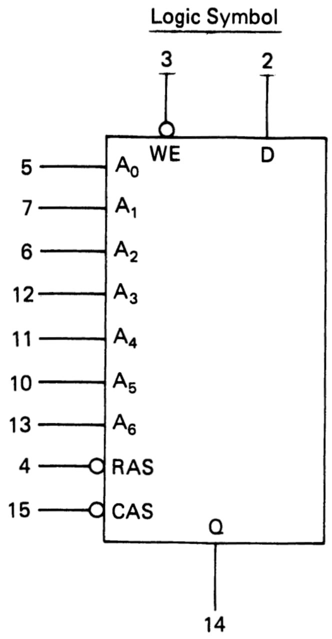
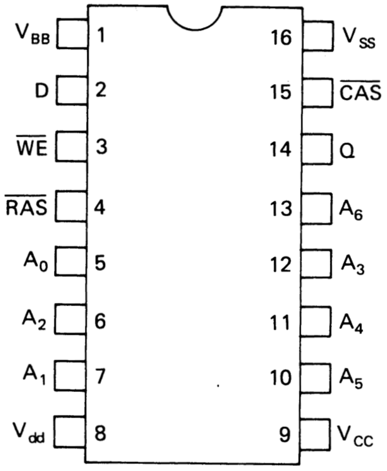


Figure 10-3. A) Block diagram for a 16K dynamic memory chip.



(B)



Pin Names

A_0–A_6	Address Inputs
D	Data Input
$\overline{WE}$	Write Enable Input (Active LOW)
$\overline{RAS}$	Row Address Strobe Input (Active LOW Clock)
$\overline{CAS}$	Column Address Strobe Input (Active LOW Clock)
Q	Data Output
V_{CC}	+5 V Power Supply
V_{SS}	0 V Power Supply
V_{BB}	−5 V Power Supply
V_{dd}	+12 V Power Supply

(C)

Figure 10-3 (continued). B) circuit symbol, C) IC package

Each cell in any given row is connected to its own vertical column (or bit line) that serves to connect it to a sense amplifier (Figure 10-4).

The DRAM read cycle is shown in Figure 10-5. The operation of $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ with respect to the address data passed to the 4116 is shown. The *write enable* ($\overline{\text{WE}}$) is an active-LOW input that must be kept HIGH during the read operation. After all of these timing actions take place, the data out line will contain a valid data signal.

The refresh cycle for the 4116 is shown in Figure 10-6. The arrangement of this chip allows us to refresh all cells using only the row address line. Either the CPU (in the case of the Z-80), the program (in 6502 and almost every other microprocessor), or an in-memory computer will supply a row address to A0 through A6 at the same time an $\overline{\text{RAS}}$ signal is generated. During this period, the data out line is open (i.e., tri-stated). This process must be accomplished not less often than every two milliseconds.

Read Only Memory (ROM) Devices

The read only memory is a semiconductor device that will store a program or data, and may be treated in the circuit as if it were a semiconductor RAM device. The difference, of course, is that the ROM will not accept data from the CPU during write operations—it allows only read operations (hence the name). The write only memory (WOM) is a joke that made the rounds of the microcomputer/semiconductor industries a few years ago and referred to an imaginary device that will accept data and never give it up again. Of course, an open conductor accomplishes the same neat trick!

Several different types of ROM are on the market. Some are permanently programmed and cannot be reprogrammed. These devices use internal fuse links that are either left intact or blown with a high current input from the external world. In one condition, the internal transistor is made LOW, while in the other, the transistor is HIGH. Another type of ROM is the erasable programmable read only memory (EPROM). This device is programmed in a manner similar to the other type, except the internal mechanism is different and allows the device to be reprogrammed. There is a quartz window in the top of the IC package that allows the chip to be exposed to an ultraviolet light source that will erase (set to HIGH) the EPROM.

ADDRESS BLOCK DECODING

Most microcomputers use more than 1K of memory, yet many of the memory chips available are only 1024-byte (with some being 256-byte).

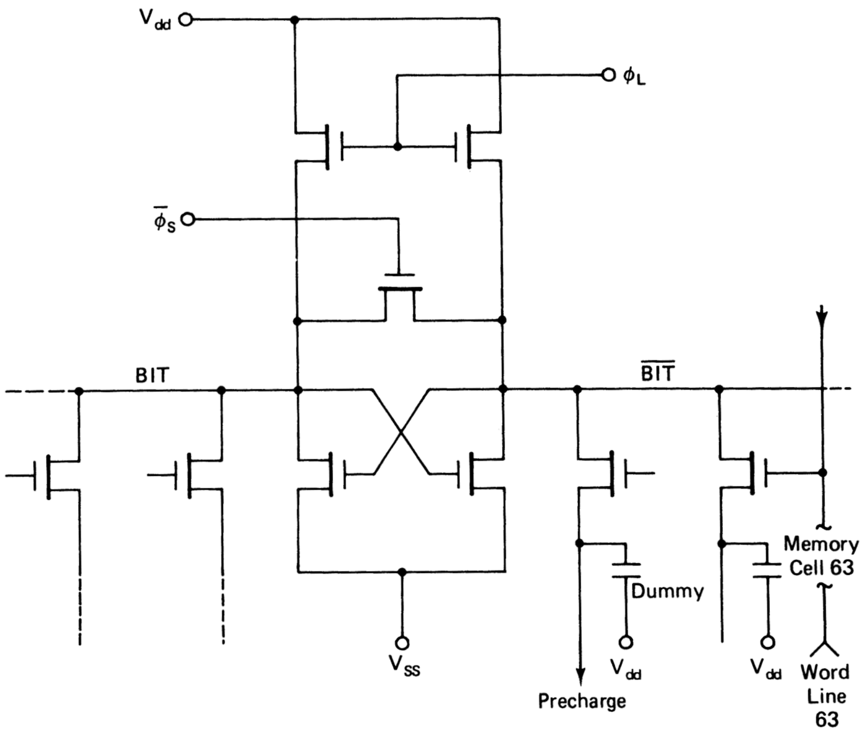


Figure 10-4. Simplified internal circuitry

Although there are more modern devices capable of very large byte arrays, many users still prefer the older, smaller devices. The question arises, "How does the memory device allocated to a location greater than the maximum address in each individual chip know when it is being addressed?" The solution seems to be ordering of the memory in 1K blocks, and then the use of some form of address decoding to tell which 1K block is being designated.

Figure 10-7 shows a selection scheme used by several manufacturers of 8K memory banks. Each block of this memory is an array of 1024 bytes, so every location can be addressed by bits A0–A9 of the address bus. The address pins for all devices are connected together to form the address bus (A0–A9). We must, however, select which of the eight blocks is addressed at any given time. One way to do this is to use a data selector IC. The 7442 device shown in Figure 10-7 is a BCD-to-1-of-10 decoder. It will examine a 4-bit binary (BCD) input word, and issue an output condition that indicates the value of that word. In this simplified example, we are going to limit the memory

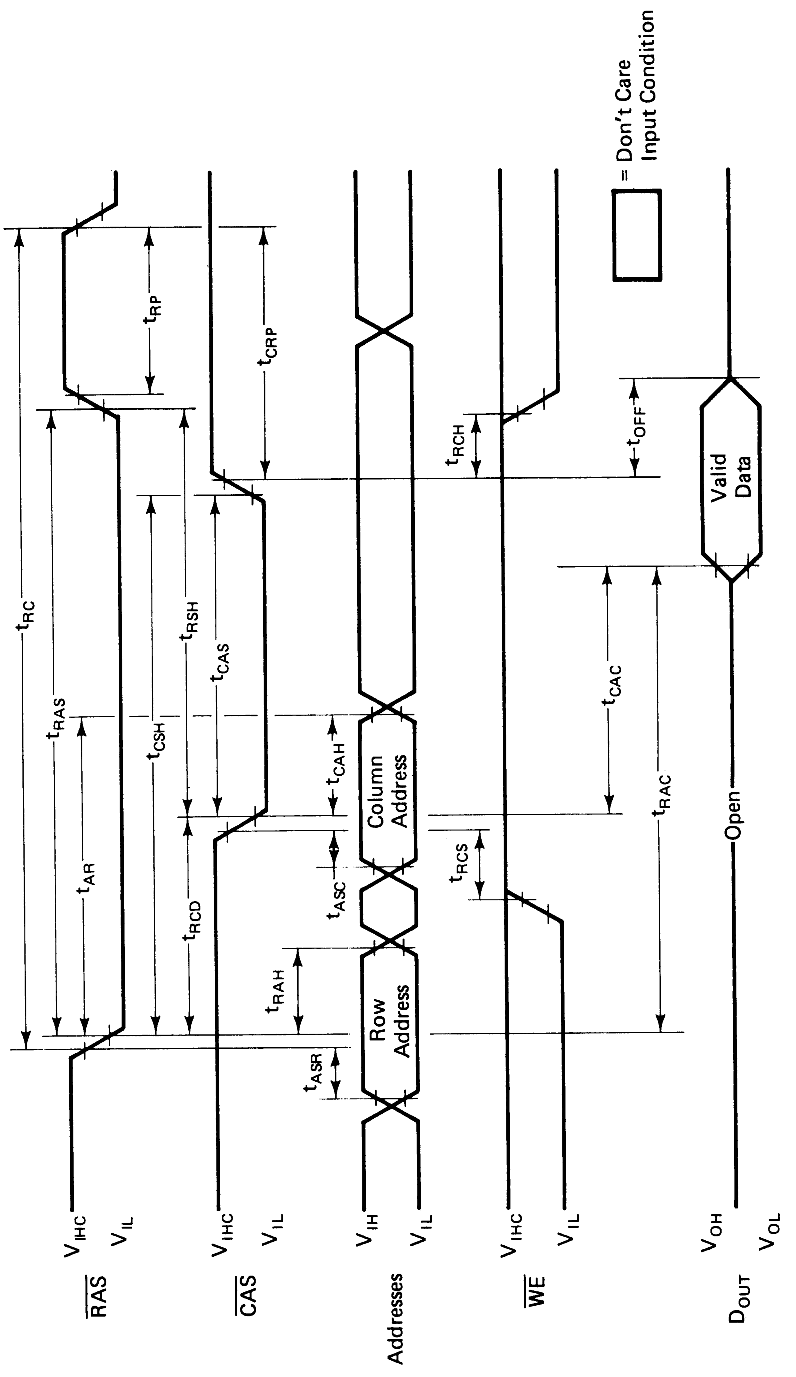


Figure 10-5. Timing diagram for 4116 device

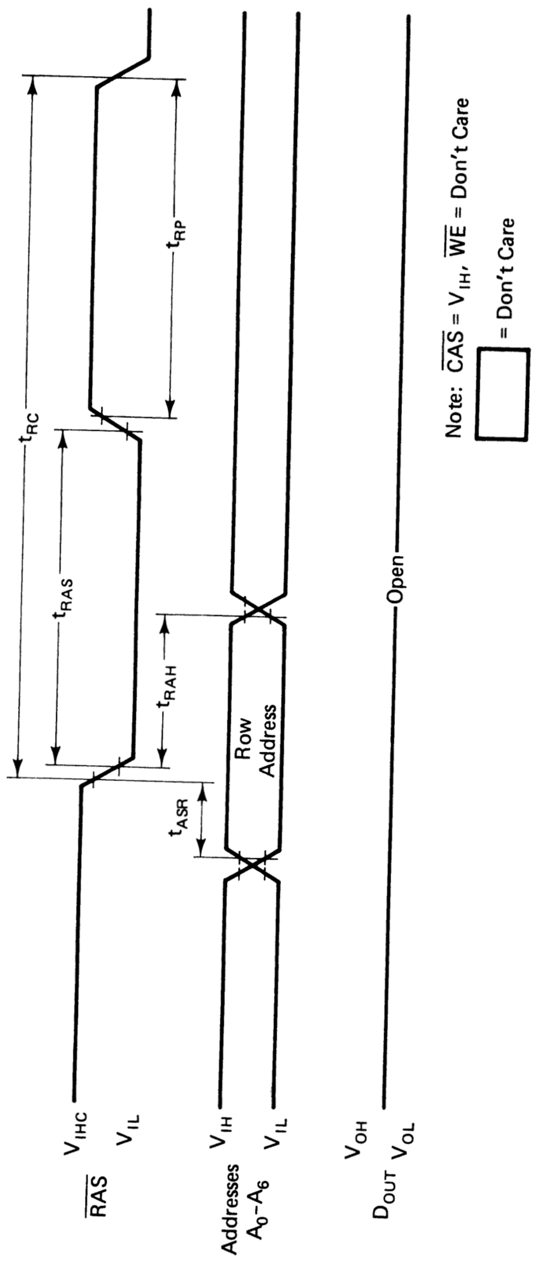
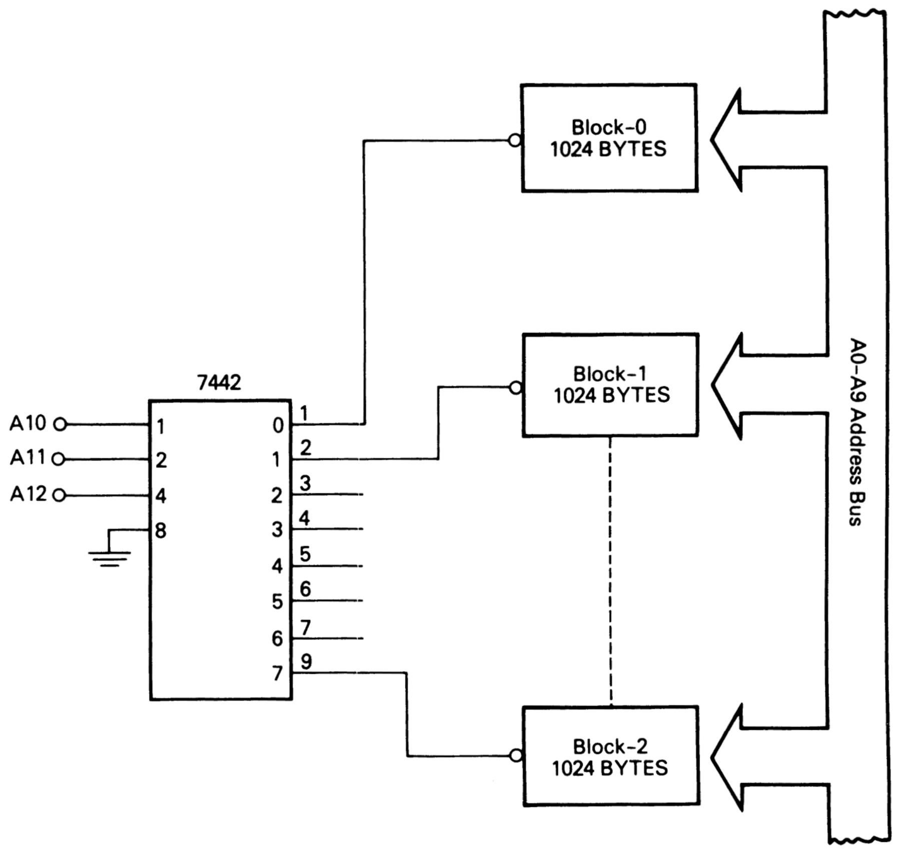


Figure 10-6. Refresh timing



(a)

		A13	A12	A11	A10
Block-Ø	0–1K	Ø	Ø	Ø	Ø
	1K–2K	Ø	Ø	Ø	1
	2K–3K	Ø	Ø	1	Ø
	3K–4K	Ø	Ø	1	1
	4K–5K	Ø	1	Ø	Ø
	5K–6K	Ø	1	Ø	1
	6K–7K	Ø	1	1	Ø
	7K–8K	Ø	1	1	1

(b)

Figure 10-7. Using 7442 in bank selection of memory; code for above

size to 8K, so only the 1, 2, and 4 inputs of the 7442 are needed. The input weighted 8 is grounded (set = 0). The 7442 indicates the active output by going LOW, exactly the right condition for the RAM devices in the memory blocks. The code that will exist on the A10–A12 bits of the address bus for the various memory addresses in the range 0–8K is shown here:

Memory Locs.	*A13*	*A12*	*A11*	*A10*	*Block*	*7442 Output*	*7442 Pin*
0K–1K	0	0	0	0	0	0	1
1K–2K	0	0	0	1	1	1	2
2K–3K	0	0	1	0	2	2	3
3K–4K	0	0	1	1	3	3	4
4K–5K	0	1	0	0	4	4	5
5K–6K	0	1	0	1	5	5	6
6K–7K	0	1	1	0	6	6	7
7K–8K	0	1	1	1	7	7	9

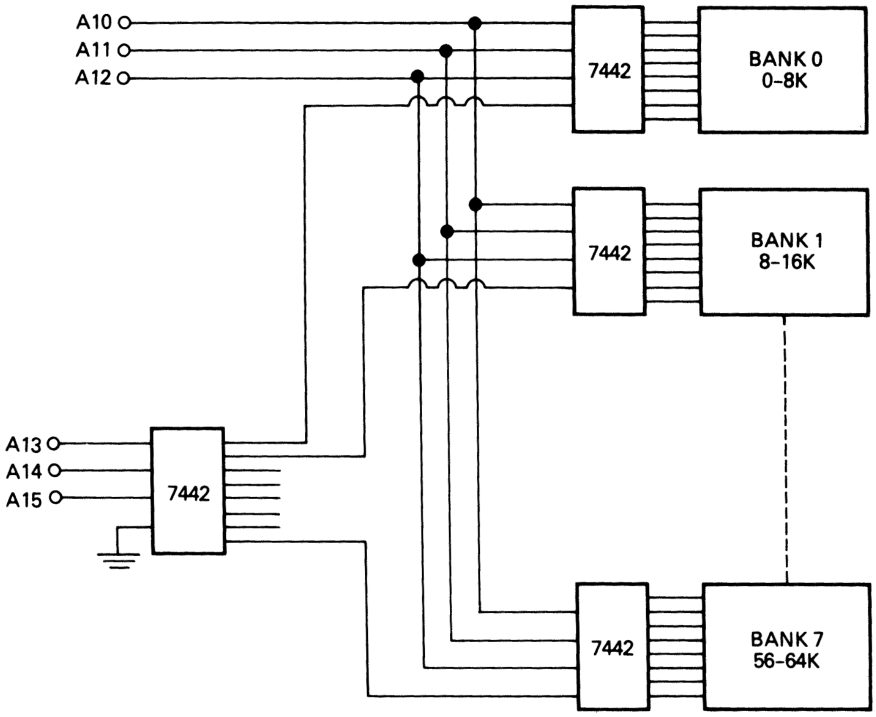


Figure 10-8. Multiple bank selection

For an 8K memory, then, the lower 10 bits of the address bus (A0–A9) select which location in the individual chips is wanted, and A10–A12 select which block of 1024 bytes contains the address.

In Figure 10-7 we limited the memory size to 8K. This was done intentionally to keep the circuit simple. But how do we select memory in ranges higher than 8K? The answer is to use the 7442 input weighted "8" as a bank select control. Recall from Figure 10-7 that this input was kept grounded. If it is HIGH, then none of the eight outputs of the 7442 will go LOW. But if it is LOW, then the circuit will work. Figure 10-8 shows a simplified selection scheme for all 65K, using the "8" weighted inputs of the 7442 block selectors as a *bank select* terminal. Each bank of 8K contains its own block select 7442, and one additional 7442 is used to select the bank of 8K that will become active. The codes existing on address lines A13–A15 for each 8K bank of locations are:

Memory Locs.	*Bank*	*A15*	*A14*	*A13*	*7442 Output*	*7442 Pin Low*
0K–8K	0	0	0	0	0	1
8K–16K	1	0	0	1	1	2
16K–24K	2	0	1	0	2	3
24K–32K	3	0	1	1	3	4
32K–40K	4	1	0	0	4	5
40K–48K	5	1	0	1	5	6
48K–56K	6	1	1	0	6	7
56K–64K	7	1	1	1	7	9

11

Interfacing I/O Devices to the 6502

The topic of input and output devices, components, and circuits is often overlooked by texts on microcomputers because I/O devices are not quite as exotic and interesting as the details of some of the microprocessor chips. But the I/O section of the computer is vitally important to the overall functioning of the machine because it determines how data are transferred in and out of the machine. In other words, the utility of the device is often determined, or more often limited, by the structure of the I/O circuitry used. After you purchase a microcomputer and decide to expand its capability, it is almost inevitable that the question of I/O ports will come up: there will probably be too few to support the extra peripherals and devices that you want to add!

The input and output functions are operated by the control signals of the microcomputer, and may take either of two forms: (1) direct I/O and (2) memory-mapped. Some microprocessor chips provide for direct I/O in the form of I/O instructions; the Z-80 is one such machine (see *Z-80 Users Manual* by Joseph J. Carr, Reston Publishing Co.). In the Z-80 device, the address of the port will be passed over the low order 8 bits (A0–A7) of the address bus, while the data from the accumulator are passed simultaneously over both the data bus (DB0–DB7) and the high order 8 bits of the address bus (A8–A15). The 8-bit memory address will support up to 256 different I/O ports that can be numbered 0 through 255. The Z-80 device control signals allow for I/O operations and are combined to produce unique IN and OUT commands to the I/O devices.

Other microprocessor chips, such as the 6502, do not provide input and output commands in the instruction set, so will not have the control signals and capabilities for direct I/O. In 6502-based machines, the input and output ports are treated as if they were memory locations: Such ports are called memory-mapped I/O ports.

While admitting that the I/O is not necessarily the most interesting aspect of microprocessor technology, we must study some of these mundane details to understand how the microcomputer deals with the outside world. To begin this study we will consider some elementary digital electronics theory and some of the devices used to form I/O ports. From an understanding of these topics you should be able to progress to designing I/O ports and interfacing techniques for the 6502.

LOGIC FAMILIES

Digital electronic circuits use assorted logic blocks, called gates (AND, OR, NOT, NAND, NOR, XOR, etc.), and flip-flops to perform the various circuit functions. On initial inspection, it seems that digital logic circuit design is made simpler because all of the logic blocks are available in integrated circuit form and can be easily connected together with impunity. The reason why this situation exists is that the IC logic devices are part of various families of similar devices. A digital logic family will use standardized input and output circuits that are designed to work with each other, use the same voltage levels for both power supply and logical signals, and generally use the same technology in construction of the devices. Common logic families in current use are TTL, CMOS, NMOS, PMOS, and MOS, with subgroups within each. Obsolete forms, such as RTL and DTL, although interesting to the owner of older equipment, are of too little interest to justify inclusion here. Also certain devices will mix technologies, e.g., an NMOS microprocessor chip that uses TTL input and output circuits to gain some of the advantages of both families.

TTL (Transistor-Transistor-Logic)

Transistor-transistor-logic (TTL, also called T^2L) is probably the oldest of the currently used IC logic families and is based on bipolar transistor technology. The bipolar transistors are the ordinary PNP and NPN types, as distinguished from the field effect transistors.

The TTL logic family uses power supply potentials of 0 and +5 volts DC, and the +5-volt potential must be regulated for proper operation of the device. Most specifications for TTL devices require the voltage to be between +4.5 VDC and 5.2 VDC, although there appear to be practical limitations on even these values. Some complex function ICs, for example, will not operate properly at potentials below +4.75 volts, despite the manufacturer's protestations to the contrary. Also, at potentials above 5.0 volts, even though less than the +5.2-volt maximum potential is "allowed," there is an excess failure rate probably due to the higher temperatures generated inside the ICs. The best rule is to keep the potential of the power supply between +4.75 and +5.0 volts; furthermore, the potential must be regulated.

Figure 11-1 shows the voltage levels used in the TTL family of devices to represent logical-1 and logical-0. The logical-1, or HIGH, condition is represented by a potential of +2.4 volts or more (+5 volts maximum). The device must be capable of recognizing any input potential over +2.4 volts as a HIGH condition. The logical-0, or LOW, condition is supposedly zero volts but most TTL devices define any potential from 0 to 0.8 volt as logical-0. The voltage region between +0.8 volt and +2.4 volts is undefined; the operation of a TTL device in this region is not predictable. Some care must be exercised to keep the TTL logical signals outside the undefined zone—a source of problems in some circuits that are not properly designed.

The inverter, or NOT gate, is the simplest form of digital logic element and contains all of the essential elements required to discuss the characteristics of the family. Figure 11-2A shows the internal circuit of a typical TTL inverter. The output circuit consists of a pair of NPN transistors connected in the "totem pole" configuration in which the transistors form a series circuit across the power supply. The output terminal is taken at the junction between the two transistors.

The HIGH state on the output terminal will find transistor Q4 turned off and Q3 turned on. The output terminal sees a low impedance (approximately 130 ohms) to the +5-volt line. In the LOW output

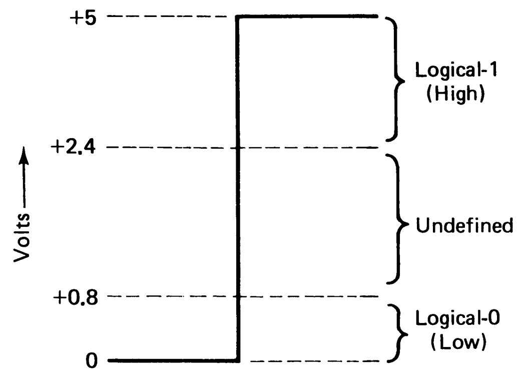


Figure 11-1. TTL logic levels

state, exactly the opposite situation exists: Q4 is turned on and Q3 is turned off. In that condition, the output terminal sees a very low impedance to ground.

The input terminal of the TTL inverter is a transistor emitter (Q1). When the input is LOW, the emitter of Q1 is grounded. The transistor is forward biased by resistor R1 so the collector of Q1 is made LOW also. This condition causes transistor Q2 to be turned off, so the voltage on its emitter is zero and the voltage on its collector is HIGH. In this situation, we have the conditions required for a HIGH output: Q4 is turned off and Q3 is forward biased, thereby connecting the output terminal through the 130-ohm resistor to the +5-volt DC power supply terminal.

Exactly the opposite situation obtains when the input terminal is HIGH. In that case, we find transistor Q1 turned off and the voltage applied to the base of Q2 HIGH. Under this condition, the collector voltage of Q2 drops and its emitter voltage rises. Transistor Q4 is turned on, grounding the output terminal, and transistor Q3 is turned off. In other words, a HIGH on the input terminal produces a LOW on the output terminal.

Figure 11-2B shows the current path when two TTL devices are connected together in cascade. The emitter of Device A input is connected to the output terminal of Device B. The input of a TTL device is a current source that provides 1.6 milliamperes at TTL voltage levels. The output transistors are capable of sinking up to 16 milliamperes. Therefore, we may conclude, for regular TTL devices, the output terminal will provide current sinking capability to accommodate up to 10 TTL input loads. Some special "buffer" devices will accommodate up to 30 TTL input loads.

The input and output capabilities of TTL devices are generally defined in terms of *fan-in* and *fan-out*. The fan-in is standardized in a unit, or standard, input load rather than current and voltage levels. This convention allows us to interconnect TTL devices simply without being concerned with matters such as impedance matching. In interfacing TTL devices it is merely necessary to ensure that the number of TTL input loads does not exceed the fan-out of the driving device. In brief, the fan-in is one unit TTL input load, while the fan-out is the output capacity expressed in the number of standard input loads that a device will drive. In the case of the regular TTL devices, the output current capacity is 16 mA, while the standard input load is 1.6 mA, so a fan-out of 16/1.6, or 10, exists.

Asking a TTL device to drive a number of TTL loads in excess of the rated fan-out will result in reduced noise margin and the possibility that the logic levels will be insufficient to reliably drive the

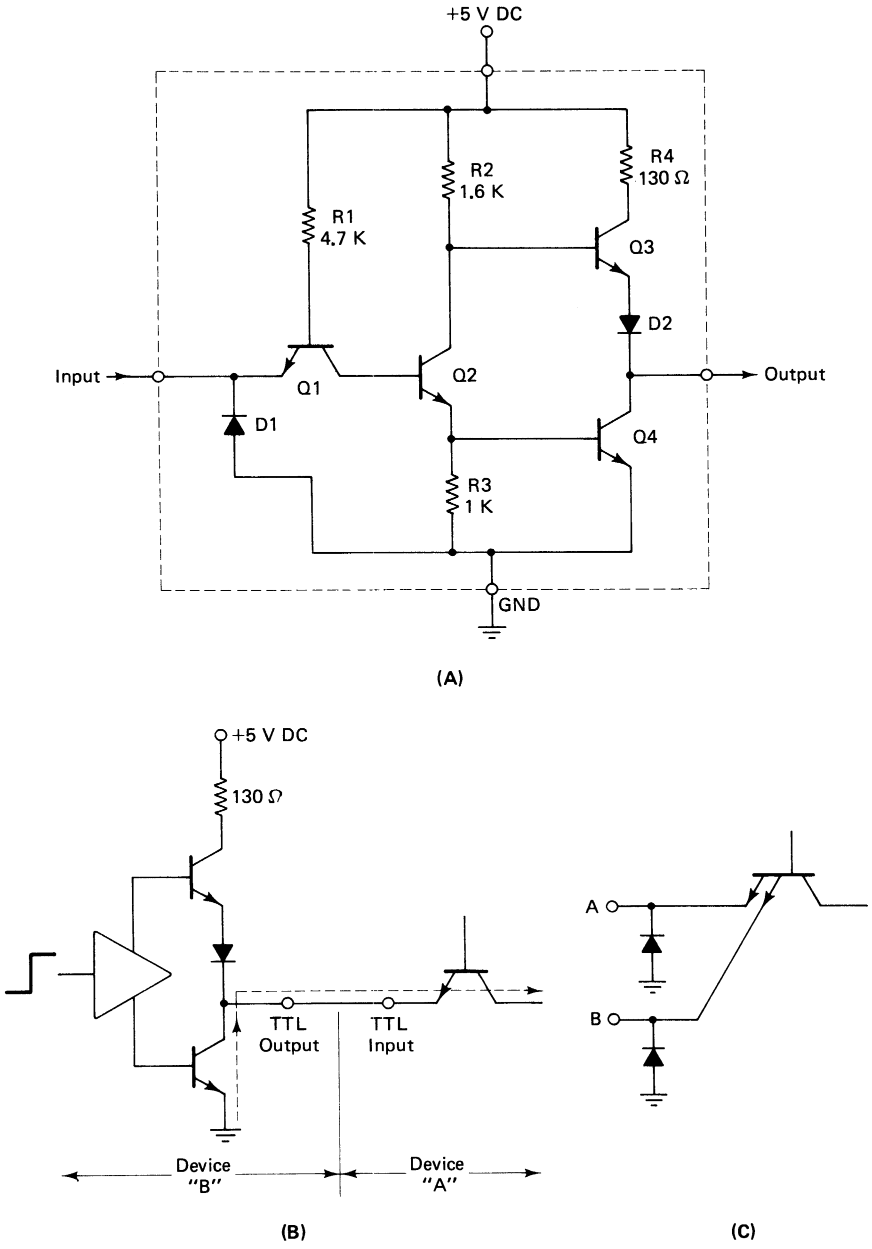


Figure 11-2. A) TTL inverter, B) TTL interface, C) multiple-emitter inputs for TTL gate.

inputs connected to the output. Some devices will provide a fan-out margin but most will not. When it is necessary to drive a large number of TTL loads, it is wise to use a high fan-out buffer.

Multiple TTL inputs are formed by adding extra emitters to the input transistor (see Figure 11-2B). This type of circuit is used on multiple input devices such as NAND gates, etc.; each emitter is capable of sourcing 1.6 mA of current and represents a fan-in of one standard TTL load.

Open-Collector Output. The standard TTL output circuit shown in Figure 11-2A must be connected to a standard TTL input in order to work properly. At times, however, it becomes necessary to interface the TTL device with some other type of device than TTL. In some cases, the external load will be at the same voltage level as TTL, but in others the voltage level might be considerably higher than +5 volts. The open-collector circuit of Figure 11-3 will accommodate such loads.

Figure 11-3 shows only the output stage of the open-collector device; all the other circuitry will be as in Figure 11-2A. Transistor Q1 is arranged so that its collector is brought out to the output terminal of the device. Since there is no current path to the V+ terminal of the power supply, an external load must be provided for the device to work. In the case of the situation shown, an external "pull-up resistor" is connected between the output terminal (i.e., Q1 collector) and +5 volts DC; for most TTL open-collector devices the value of the pull-up resistor is 2 kohm to 4 kohms. Other loads and higher voltages can be accommodated, provided that the DC resistance of

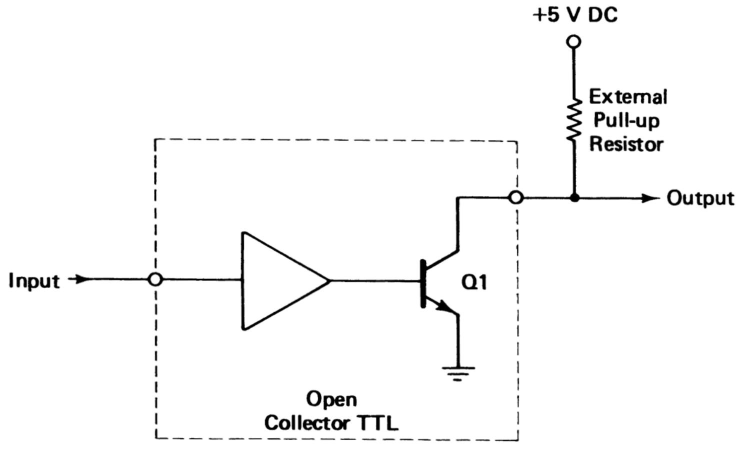


Figure 11-3. Open-collector TTL devices

the load is sufficient to keep the collector current in Q1 within specified limits.

Speed vs. Power. The TTL logic family is known for its relatively fast operating speeds. Most devices will operate to 18–20 MHz, and some selected devices operate to well over 30 MHz. But the operating speed is not without a concomitant trade-off: increased operating power. Unfortunately, higher speed means higher power dissipation. The problem is the internal resistances and capacitances of the devices. The operating speed is set in part by the RC time constants of the internal circuitry. To reduce the time constant and thereby increase the operating speed, it is necessary to reduce the resistances and that will necessarily increase the current drain and power consumption.

TTL Nomenclature. Each logic family uses a unique series of type numbers for the member devices so that users can identify the technology being used from the number. With very few "house number" exceptions, TTL type numbers will have either four or five digits beginning with the numbers 54 or 74. The normal devices found most commonly are numbered in the 74xx and 74xxx series, while the higher grade "military specification" devices carry 54xx and 54xxx numbers. The 54 and 74 series retain the same "xx" or "xxx" suffix for identical devices. For example, the popular NOR gate will be numbered 7402 in commercial grade components and 5402 in military grade. In general use, we can substitute the more reliable 54xx devices for the identical 74xx devices.

TTL Subfamilies. Certain specialized TTL devices are used for certain purposes, such as increased operating speed, lower power consumption, etc. These family subgroups include (in addition to regular TTL) low power (74Lxx), high speed (74Hxx), Schottky (74Sxx), and low power Schottky (74LSxx) devices. A principal difference between these groups that must be addressed by the circuit designer or interfacer is the input and output current requirements. In most cases, the following levels apply:

Subfamily	*Input Current*	*Output Current*
74xx	1.6 mA	16 mA
74Lxx	0.18 mA	3.6 mA
74Hxx	2.0 mA	20 mA
74Sxx	2.0 mA	20 mA
74LSxx	0.4 mA	8.0 mA

CMOS (Complementary Metal Oxide Semiconductor)

The complementary metal oxide semiconductor, or CMOS, digital IC logic family is based on the metal oxide semiconductor field effect transistor (MOSFET). In general, CMOS devices are slower in operating speed than TTL devices, but have one immensely valuable property: low power dissipation. The CMOS device presents a high impedance across the DC power supply at all times, except when the output is undergoing transition from one state to the other. At all other times, the CMOS device draws only a few microamperes of electrical current, making it an excellent choice for large systems where speed of operation is not the most important specification.

Figure 11-4 shows two CMOS devices which are at least representative of the large family of related logic elements. Figure 11-4A illustrates a simple CMOS inverter. Note that it consists of N-channel and a P-channel MOSFETs connected such that their respective source-drain paths are in series, while the gate terminals are in parallel. This arrangement is reminiscent of push-pull operation because the N-channel and P-channel devices turn on and off with opposite polarity signals. As a result, one of these two transistors will have a low channel resistance with the input LOW, while the other will offer a very high resistance (megohms). When the input is made HIGH, then the role of the two transistors is reversed: the one with the low channel resistance becomes high resistance, while that with the high resistance goes LOW. This operation has the effect of connecting the output terminal to either V_{dd} or V_{ss} depending upon whether the input is HIGH or LOW. Since, in both cases, one of the series pair is high resistance, the total resistance across the V_{dd}-V_{ss} power supply is HIGH. Only during the transistion period, when both transistors have a medium range source-drain resistance, will there be any appreciable load in the power supply. The output terminal will not deliver any current because it will be connected to another CMOS input, which has a very high impedance. As a result, there is never any time when the CMOS IC, operated only in conjunction with other CMOS devices, will draw any appreciable current. An example of the difference between TTL and CMOS current levels is seen by comparing the specifications for a common quad 2-input NAND gate in both families. The TTL version needs 25 milliamperes, while the CMOS device requires only 15 microamperes.

Figure 11-4B shows a typical CMOS AND gate. The two inputs are connected to independent inputs of a pair of series-connected N-channel MOSFETs. The output of this stage will not change state unless both inputs are active, a result of the series connection.

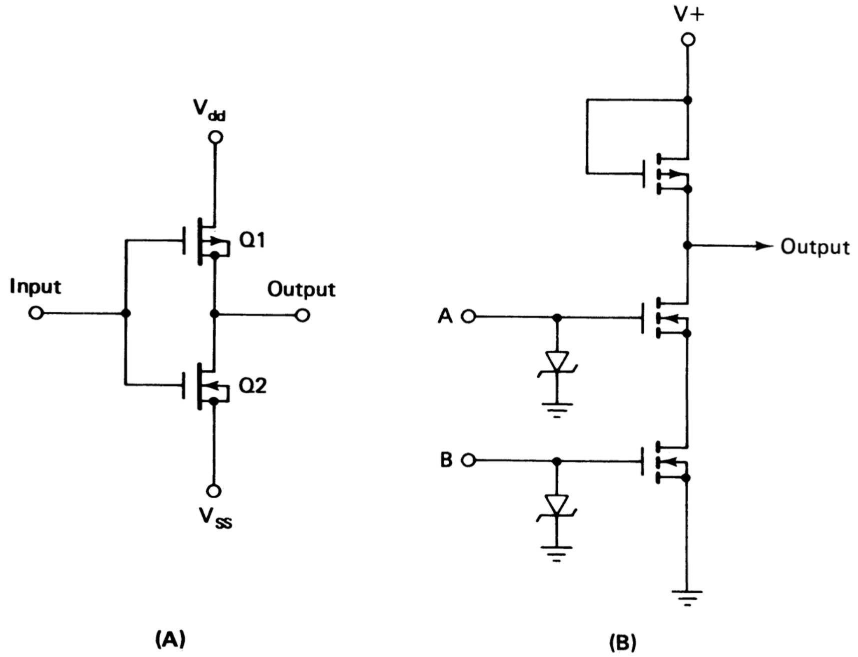


Figure 11-4. A) CMOS inverter circuit, B) CMOS gate

The operating speed of typical CMOS devices is limited to 4–5 MHz, although some 10–15 MHz devices are known. The speed is the principal disadvantage to the CMOS line; typical TTL devices operate to 20 MHz but require more current.

Another problem with the CMOS device is sensitivity to static electricity. The typically very thin insulating layer of oxide between the gate element and the channel has a breakdown voltage of 80 to 100 volts. Static electricity, on the other hand, can easily reach values of 1,000 volts or more! Whenever the static is sufficient to cause a biting spark when you touch a grounded object, you can bet that it was generated by a potential of 1,000 volts or more! This potential is sufficient to destroy CMOS devices. This problem is especially critical in dry climates or during the low humidity portions of the year. However, methods of working with CMOS allow us to minimize damage to the device. In general, the CMOS working rules require use of a grounded working environment, grounded tools, and that we avoid wearing certain wool or artificial fiber garments. Also, the B series (e.g., CA-4001B) have built-in zener diodes to protect the delicate gate structure by shunting dangerous potentials around the gate.

Tri-state Devices

Ordinary digital IC logic devices are allowed only two permissible output states: HIGH and LOW, corresponding to TRUE/FALSE logic or 1/0 of the binary system. In the HIGH state, the output is typically connected through a low impedance to a positive power supply, while in the LOW state the output is connected to either a negative power supply or ground. While this arrangement is sufficient for ordinary digital circuits, a problem exists when two or more outputs are connected together but must operate separately. Such a situation exists in a microcomputer on the data bus. If any one device on the bus stays LOW, then it more or less commands the entire bus: No other changes on any other device will be able to affect the bus so the result will be chaos. Also, even if we could conspire to make all bits HIGH when not in use, there would still be a loading factor as well as ambiguity as to which device is turned on at any given time.

The answer to the problem is in tri-state logic, as shown schematically in Figure 11-5. Tri-state devices, as the name implies, have a third permissible output state. This third state effectively disconnects the output terminal from the workings of the IC. In Figure 11-5, switch S1 represents the normal operating modes of the device. When the input is LOW, switch S1 is connected to R1 so the output would be

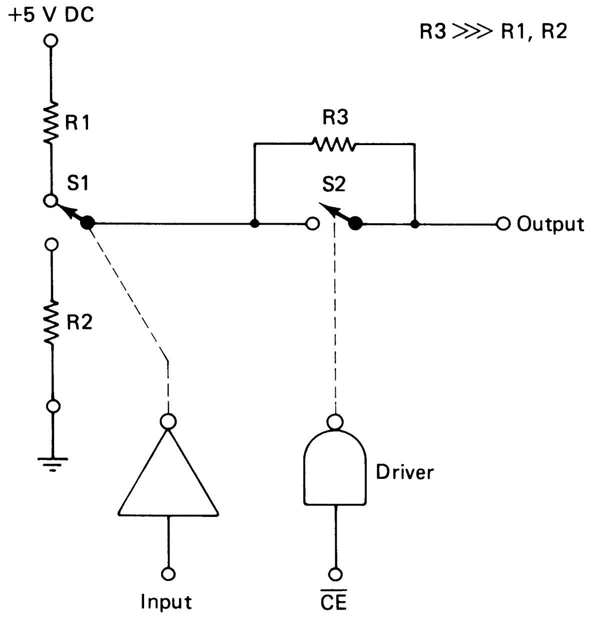


Figure 11-5. Tri-state output equivalent circuit

HIGH. Similarly, when the input is HIGH, switch S1 is connected to R2 so the output is LOW. The third state is generated by switch S2. When the active-LOW chip enable ($\overline{CE}$) terminal is made LOW, then switch S2 is closed and the output terminal is connected to the output of S1. When the $\overline{CE}$ terminal is HIGH, however, switch S2 is open so the output floats at a high impedance (represented by R3). Because of this operation, the tri-state device can be connected across a data bus line and will not load the line except when $\overline{CE}$ is made LOW.

An advantage of tri-state digital devices is that the chip enable terminals can be driven by device select pulses, creating a unique connection to the data bus that is not ambiguous to the microcomputer. In other words, the computer will "know" that only the data from the affected input port or device is on the bus whenever that $\overline{CE}$ is made LOW.

INTERFACING LOGIC FAMILIES

One of the defining characteristics of a logic family is that the inputs and outputs of the devices within the family can be interconnected with no regard to interfacing. A TTL output can always drive a TTL input, and a CMOS output can always drive a CMOS input without any external circuitry other than a conductor. But when we want to interconnect logic elements of different families, then some consideration must be given to proper methods. In some cases, it will suffice to simply connect the output of one device to the input of the other, while in other cases some external circuitry is needed.

Figure 11-6A shows a series of cascade inverters. The CMOS device is not comfortable driving the TTL input, and the TTL input is not happy with the CMOS output. As a result, we must use a special CMOS device that will behave as if it has a TTL output while retaining its CMOS input: 4050 and 4049. The 4049 device is a hex inverting buffer, while the 4050 is the same in noninverting configuration. The special character of these devices is the bipolar transistor output that will mimic the TTL output *if* the package V+ potential is limited to +5 volts DC. The 4040/4050 will operate to potentials up to +15 volts, but is TTL compatible only at a V+ potential of +5 volts DC, with the other side of the device power supply grounded. The input of the 4049/4050 is CMOS, so is compatible with all CMOS outputs.

The TTL input is a current source, so the TTL output depends for proper operation on driving a current source (naturally). The CMOS input, however, is a very high impedance because the CMOS family is voltage-driven. If we want to interface an ordinary TTL output to a CMOS input (see Figure 11-6B), then we must provide a pull-up resistor between the TTL output terminal and the +5-volt DC power supply. A value between 2 and 4 kohms is selected to make the current source mimic a TTL input current level.

The method in Figure 11-6B works well in circuits where both CMOS and TTL devices operate from a +5-volt DC power supply. While this is the usual situation in most circuits, on occasion the TTL and CMOS devices operate from different potentials; the correct interfacing method is shown in Figure 11-6C. Here we use an open-collector TTL output with a resistance to the V_{dd} power supply (used by the CMOS device) that is sufficiently high to keep the current flowing in the TTL output at a level within tolerable limits.

We can use a single 4049/4050 device to drive up to two regulator TTL inputs (Figure 11-6D), and an ordinary CMOS device will drive a single "LS" series TTL input. The 4001 and 4002 CMOS devices are capable of directly driving a single regular TTL input. With the exception of the 4049/4050 device, these methods depend upon the CMOS and TTL devices operating from a common +5-volt DC power supply. If the CMOS devices are operated at higher potentials, then all bets are off and we will be forced into using the 4049/4050 method discussed previously to prevent burnout of the TTL input.

Most microprocessor chips have limited output line capacity, most being limited to one or two TTL input loads. Most of the MOS series microprocessor chips use MOS logic internally, but have TTL-compatible output lines. In the case of a two loads output, the total allowable output current is 3.2 milliamperes. However, many TTL-compatible inputs may be connected to the data bus or address bus of the microcomputer. We need a high current bus driver on each line of the bus to accommodate these higher current requirements. Figure 11-7 shows a series of eight noninverting bus drivers interfacing the data bus of a microcomputer (DB0–DB7) with the data bus outputs of the microprocessor chip (B0–B7). This circuit will increase the drive capacity of the microcomputer from a fan-out of 2 to a fan-out of 30 or even 100, depending upon the bus driver selected.

FLIP-FLOPS

All of the gates used in digital electronics are transient devices. In other words, the output state disappears when the input stimulus disappears; the gate has no memory. A flip-flop, however, is a circuit that is capable of storing a single bit, one binary digit, or data. An array of flip-flops, called a register, can be used to store entire binary words in

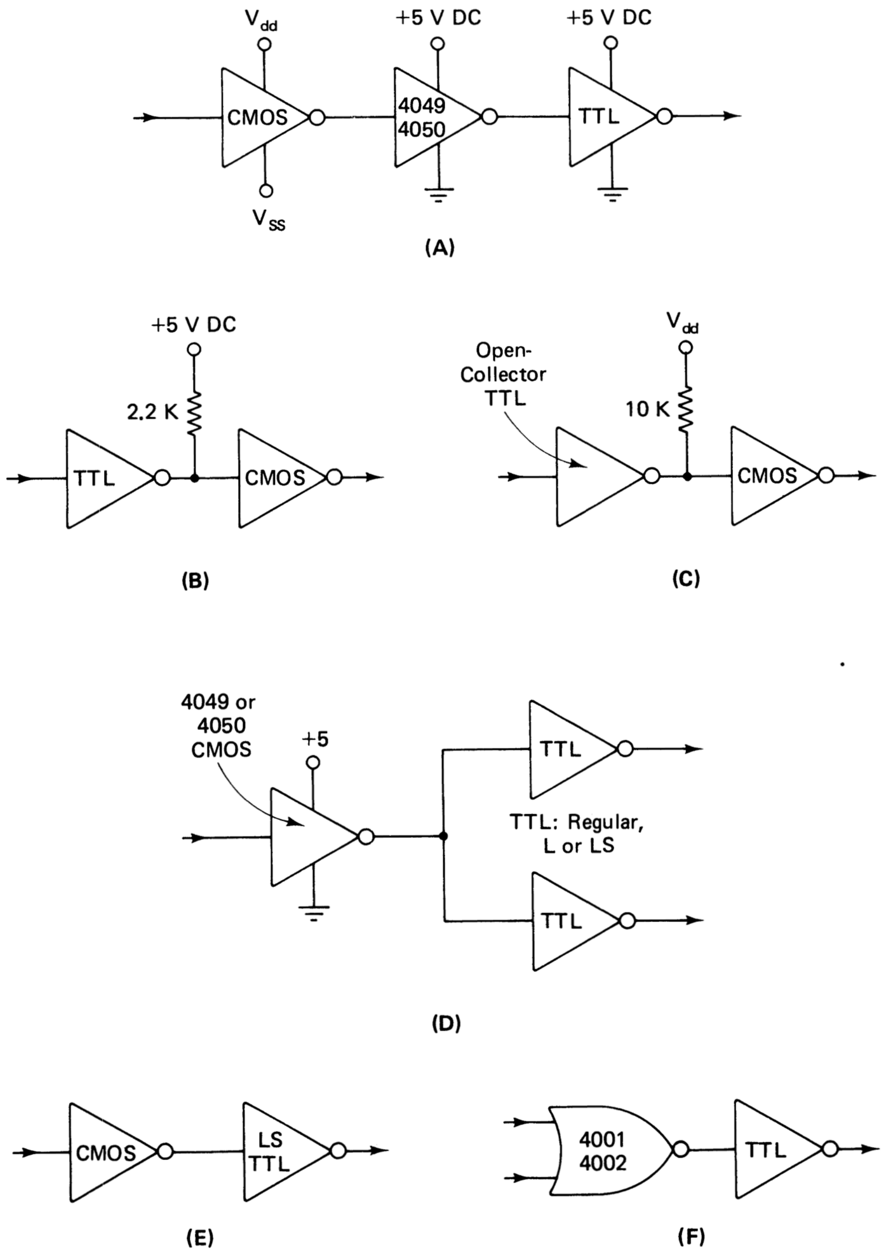


Figure 11-6. IC logic element interfacing A) any-CMOS-to-4049/4050-to-single TTL, B) TTL-to-CMOS operated from +5 VDC, C) TTL-to-CMOS other than +5 VDC, D) 4049/4050 to drive TTL, E) CMOS-to-LS TTL (i.e., 74LS series), F) 4001/4002 CMOS to TTL

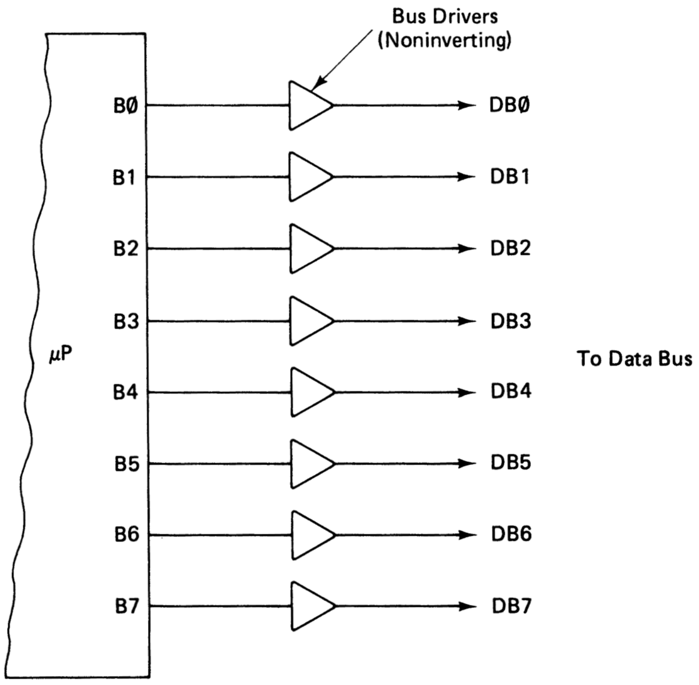


Figure 11-7. Using bus driver stages to increase the bus power capacity

the computer. All of these circuits can be built with discrete digital gates, even though few modern designers would do so because the various forms of flip-flop are available as discretes in their own right.

Figure 11-8 shows the basic reset-set, or *RS* flip-flop. The two versions are based on the NOR and NAND gates, respectively. An RS flip-flop has two inputs, *S* and *R* for *set* and *reset*. When the *S* input is momentarily made active, then the output terminals go to the state in which Q = HIGH and NOT-Q = LOW. The *R* input causes just the opposite reaction: Q = LOW and NOT-Q = HIGH. These inputs must not be made active simultaneously, or an unpredictable output state will result.

Figure 11-8A shows the RS flip-flop made from a pair of 2-input NAND gates. In each case, the output of one gate drives one input of the other; the gates are said to be cross-coupled. The alternate inputs of each gate form the input terminals of the flip-flop.

The inputs of the NAND gate version of the RS flip-flop are active-LOW. This means that a momentary LOW on either input will cause the output action. For this reason, the NAND gate version is sometimes designated an $\overline{\mathrm{RS}}$ FF, and the inputs designated $\overline{\mathrm{S}}$ and $\overline{\mathrm{R}}$, respectively.

The NOR gate version of the RS flip-flop is shown in Figure 11-8B. In this circuit, the inputs are active-HIGH so the output states change by applying a HIGH pulse momentarily. The circuit symbol for the RS flip-flop is shown in Figure 11-8C. In some instances, the NAND version will be indicated by the same circuit, while in others there will be either $\overline{\text{R}}$ and $\overline{\text{S}}$ indications for the inputs or circles indicating inversion at each input terminal.

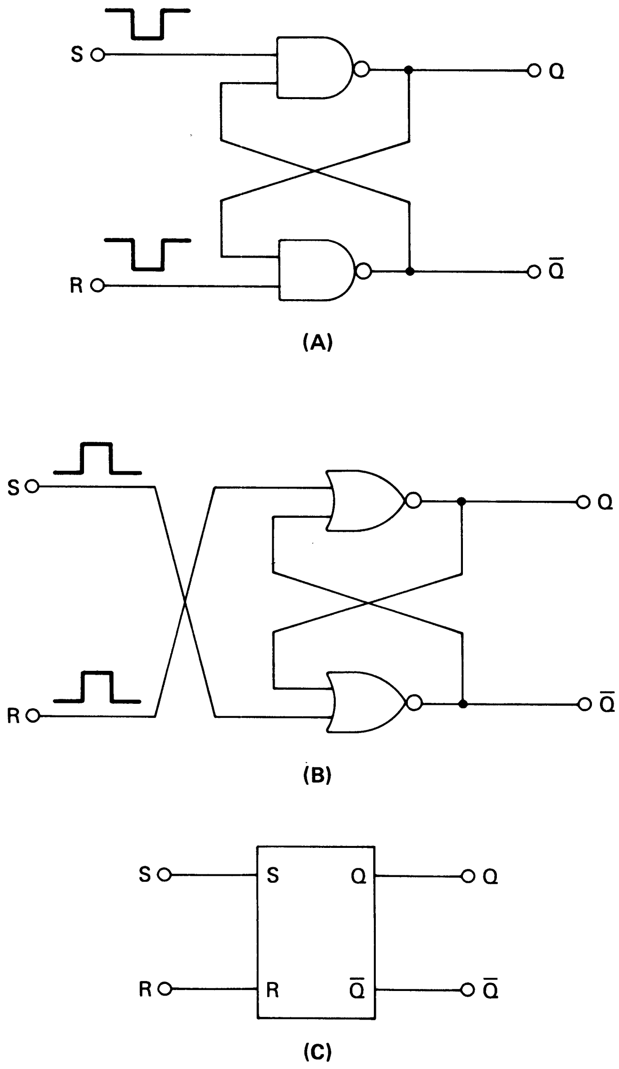


Figure 11-8. A) NAND-based RS flip-flop, B) NOR-based RS flip-flop, C) RS flip-flop circuit symbol

The RS flip-flop operates in an asynchronous manner, i.e., the outputs will change any time an appropriate input signal appears. Synchronous operation, which is required in most computer-oriented circuits, requires that output states change only coincident with a system clock pulse. The circuit in Figure 11-9 is a clocked RS flip-flop. Gates G3 and G4 form a normal NOR-based RS flip-flop. Control via a clock pulse is provided by gates G1 and G2. One input of each is connected to the clock line. These two gates will not pass the R and S pulses unless the clock line is HIGH. The input lines can change all they want between clock pulses, but an output change is effected *only* when the clock pulse is HIGH.

A Type-D flip-flop (Figure 11-10A) is made by using an inverter to ensure that the S and R inputs of a clocked RS flip-flop are always complementary. The S input of the RS flip-flop and the input of the inverter that drives the R input of the RS FF are connected in parallel. When the S input is made HIGH, therefore, the R input will be LOW. Similarly, a LOW on the S input will place a HIGH on the R input. The circuit symbol for the Type-D FF is shown in Figure 11-10B.

The rule for the operation of the Type-D flip-flop is: The input data applied to the D terminal will be transferred to the outputs *only* when the clock line is active. Figure 11-10C shows a typical timing diagram for a level-triggered Type-D flip-flop that has an active-HIGH clock. The output line of this flip-flop will follow the input line only when the clock line is HIGH. Trace D shows the data at the D-input, while trace Q shows the output data; CLK shows the clock line, which is presented with a series of regular pulses.

At time T_0, the data line goes HIGH, but the clock line is LOW, so no change will occur at output Q. At time T_1, however, the clock line goes HIGH; the data line is still HIGH so the output goes HIGH. Note that the Q output remains HIGH after pulse T1 passes, and it will continue to remain HIGH even when the data input drops LOW again. In other words, the Q output of the Type-D flip-flop will re-

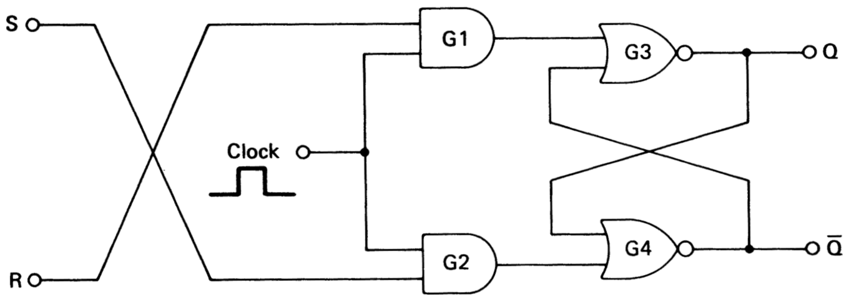


Figure 11-9. Clocked-RS flip-flop

member the last valid data present on the D input when the clock pulse went inactive. At time T_2 we find another clock pulse, but this time the D input is LOW. As a result, the Q output drops LOW. The process continues for times T_3 and T_4. Note that, in each case, the output terminal follows the data applied to the input only when the clock pulse is present!

The example shown here is for a level-triggered Type-D flip-flop, which will allow continuous output changes while the clock line is HIGH. An edge triggered Type-D flip-flop timing diagram is shown in Figure 11-10D. In this case, the data on the outputs will change only during either a rising edge of the clock pulse (positive edge triggered) or on the falling edge of the clock pulse (negative edge triggered). The flip-flop will respond only during a very narrow period of time.

I/O PORTS: DEVICES AND COMPONENTS

A number of devices on the market can be used for input and output circuitry in microcomputers. Some devices are merely ordinary TTL or CMOS digital integrated circuits that are adaptable to I/O service. Still others are special-purpose integrated circuits that were intended from the inception as I/O port devices. Most of the microprocessor chip families contain at least one general-purpose I/O companion chip that is specially designed to interface with that particular chip. In this chapter, we will study some of the more common I/O components. Keep in mind, however, that there are many other alternatives that may be better than those shown here. You are advised to keep abreast of the integrated circuits that are available from various manufacturers.

Figure 11-11 shows the TTL 74100 device. This integrated circuit is a dual 4-bit latch circuit. When we connect the latch strobe terminals together (pins 12 and 23), we find that the device is usable as an 8-bit latch. The 74100 device can be used as an output port.

The input lines of the 74100 device are connected to bits DB0 through DB7 of the data bus. The Q outputs of the 74100 are being used as the data lines to the external device. The two strobe lines are used to gate data from the data bus onto the Q outputs of the 74100. The data latch (including the 74100) will transfer data at the D inputs to the Q outputs when the strobe line is HIGH (note the similarity to the operation of the Type-D flip-flop—the data latch is a special case of the Type-D FF in which the clock line is labelled *strobe*). When the WRITE signal goes HIGH, therefore, the data on the bus is transferred to the Q outputs of the 74100. The data will remain on the Q outputs

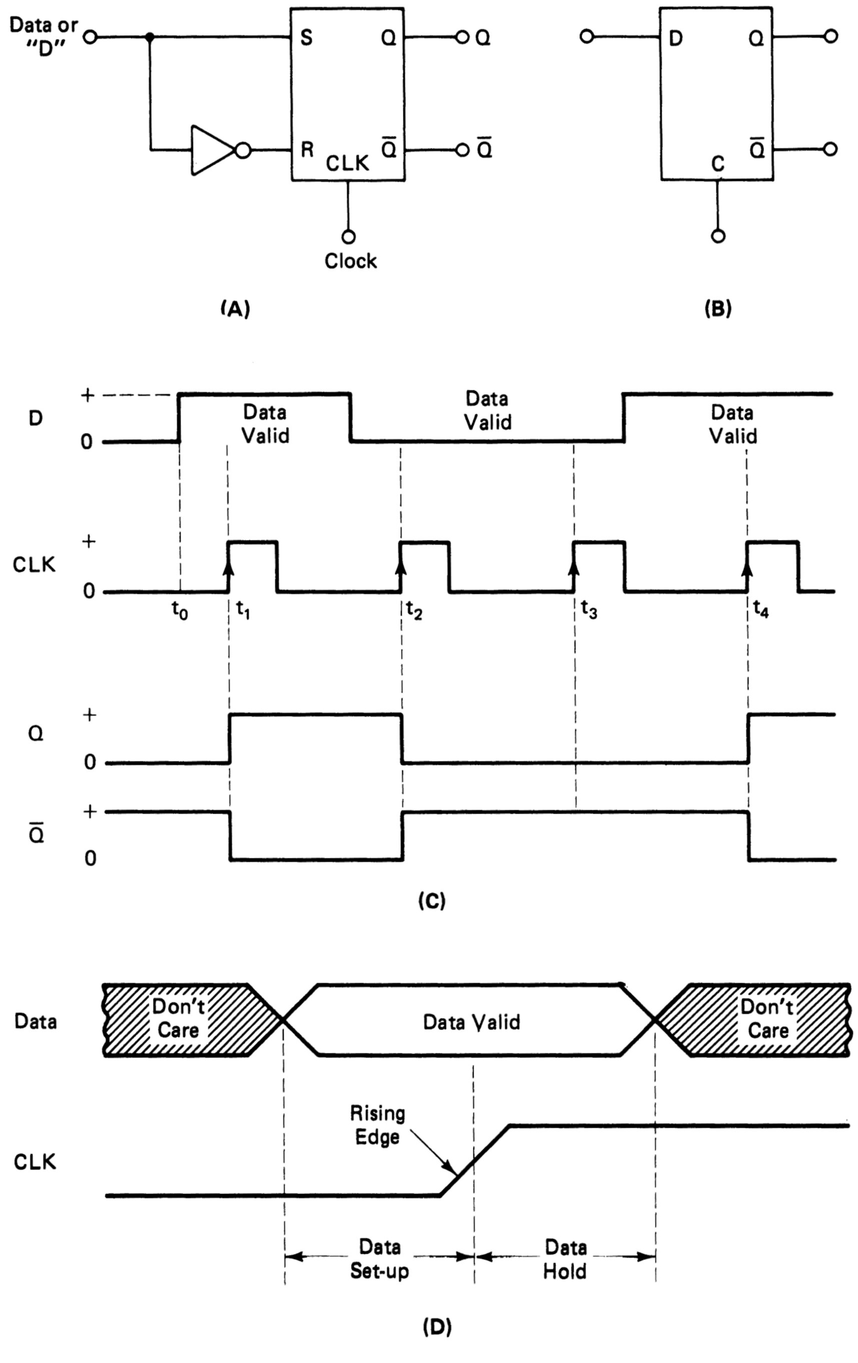


Figure 11-10. A) Type-D flip-flop made from RS flip-flop, B) circuit symbol, C) timing diagram, D) timing diagram expanded

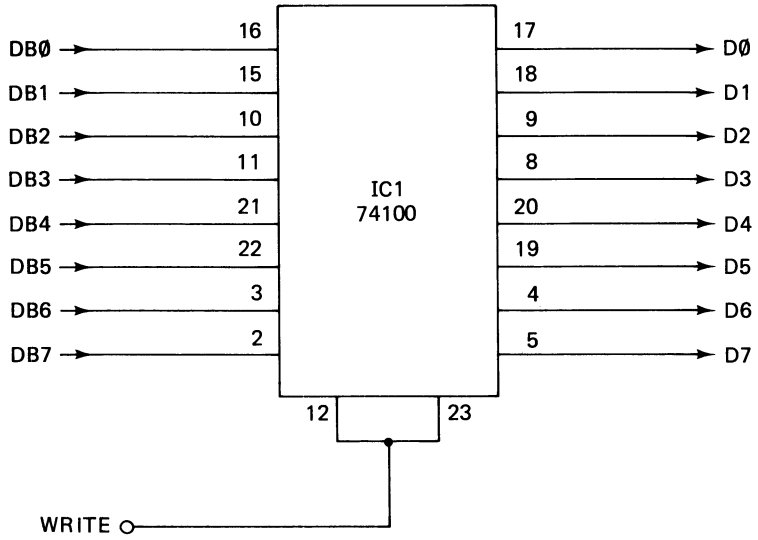


Figure 11-11. 74100 chip used as an output port

even after the WRITE signal goes LOW again. This type of output, therefore, is called a *latched output.*

It is not necessary to use a single integrated circuit for the latched output circuit. For example, we could use a pair of 7475 devices, or an array of 8 Type-D flip-flops, although one wonders why!

Input ports cannot use ordinary 2-state output devices because there may be a number of devices sharing the same data bus lines. If any one device, whether active or not, develops a short to ground then that bit will be permanently LOW, regardless of what other data are supposed to be on the line. In addition, it is possible that some other device will output a HIGH onto the permanently LOW line and thereby cause a burn-out of another IC. Similarly, a short-circuit of any given output to the V+ line will place a permanent HIGH on that line. Regardless of the case, placing a permanent data bit onto a given line of the data bus always causes a malfunction of the computer or its resident program. To keep the input ports "floating" harmlessly across the data bus lines, we must use tri-state output components for the input ports; such components were discussed earlier in this chapter (see Figure 11-5).

A number of 4-bit and 8-bit tri-state devices on the market can be used for input port duty. Figure 11-12A shows the internal block diagram for the 74125 TTL device. This device is a quad noninverting buffer with tri-state outputs. A companion device, 74126, is also useful for input port service if we want or need an inverted data signal. The

74126 device is a quad inverter with tri-state outputs. Each stage in the 74125/74126 devices has its own enable terminal ($\overline{C1}$ through $\overline{C4}$) that is active-LOW. When the enable terminal is made LOW, therefore, the stage will pass input data to the output and operate in the manner normal to TTL devices. If the enable terminal is HIGH, however, then the output floats at a high impedance so will not load the data line to which it is connected.

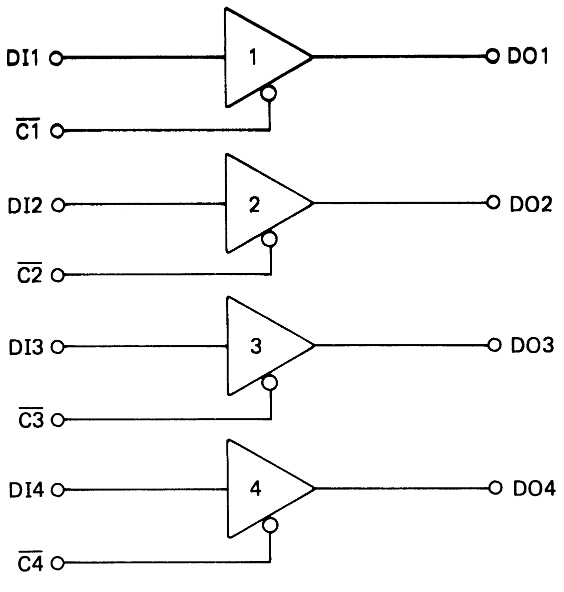
Figure 11-12B shows a pair of 74125 devices connected to form a single 8-bit input port. The output lines from each 74125 (i.e., pins 3, 6, 8, and 11) are connected to lines DB0 through DB7 of the data bus. The input pins of the 74125 (pins 2, 5, 9, and 12) are used to accept the data from the outside world.

The $\overline{\text{READ}}$ signal generated by the microprocessor chip and the device select circuits is used to turn on the 74125 devices. Note that all four enable lines of each 74125 device are parallel-connected so that all stages will turn on at the same time.

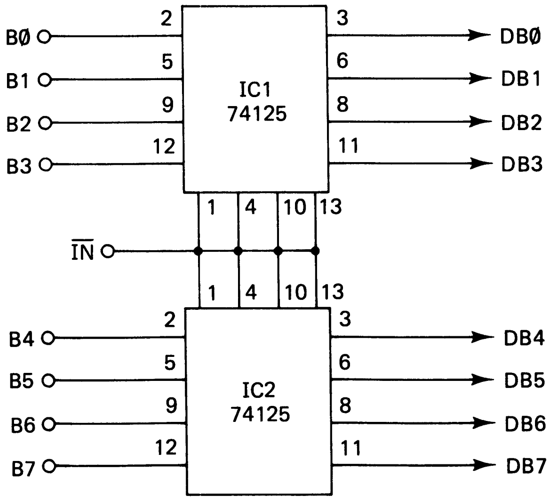
The output lines of the input port are not latched. Therefore, the data will disappear when the $\overline{\text{READ}}$ signal becomes inactive, exactly the requirements of an input port on a shared bus!

Another useful input port device is the 74LS244 TTL integrated circuit. Like the 74125 device, the 74LS244 has tri-state outputs. The 74LS244 is an array of eight noninverting buffer stages arranged in a 2-by-4 arrangement in which four devices share a common enable terminal. In Figure 11-13A, we find that stages A1 through A4 are driven by chip enable input $\overline{\text{CE1}}$ (pin 1), while B1 through B4 are driven by chip enable input $\overline{\text{CE2}}$ (pin 19). In the circuit in Figure 11-13B we strap the two chip enable terminals together to force the 74LS244 device to operate as a single 8-bit input port. The eight input lines are connected to the respective input terminals of the 74LS244, while the output lines are connected to their respective data bus lines. When the $\overline{\text{READ}}$ signal becomes active (LOW), then data on B0 through B7 will be gated onto data bus lines DB0 through DB7.

The techniques used thus far in this chapter require separate integrated circuits for input and output functions. While this is often satisfactory, it involves an excessive number of chips for some applications. We can, however, make use of combination chips in which the input and output functions are combined. Several devices on the market are classified as *bidirectional bus drivers*. These devices will pass data in either direction depending upon which is selected by the control signals. Typical devices used for several years in microcomputer designs are the 4-bit 8216/8226 devices and the 8-bit 8212 device, all by Intel. Originally, these devices were intended for use in the 8080A microprocessor circuit. Even though the 8080A has been long since



(A)



(B)

Figure 11-12. A) internal circuit of 75125 chip, B) 74125 as an input port

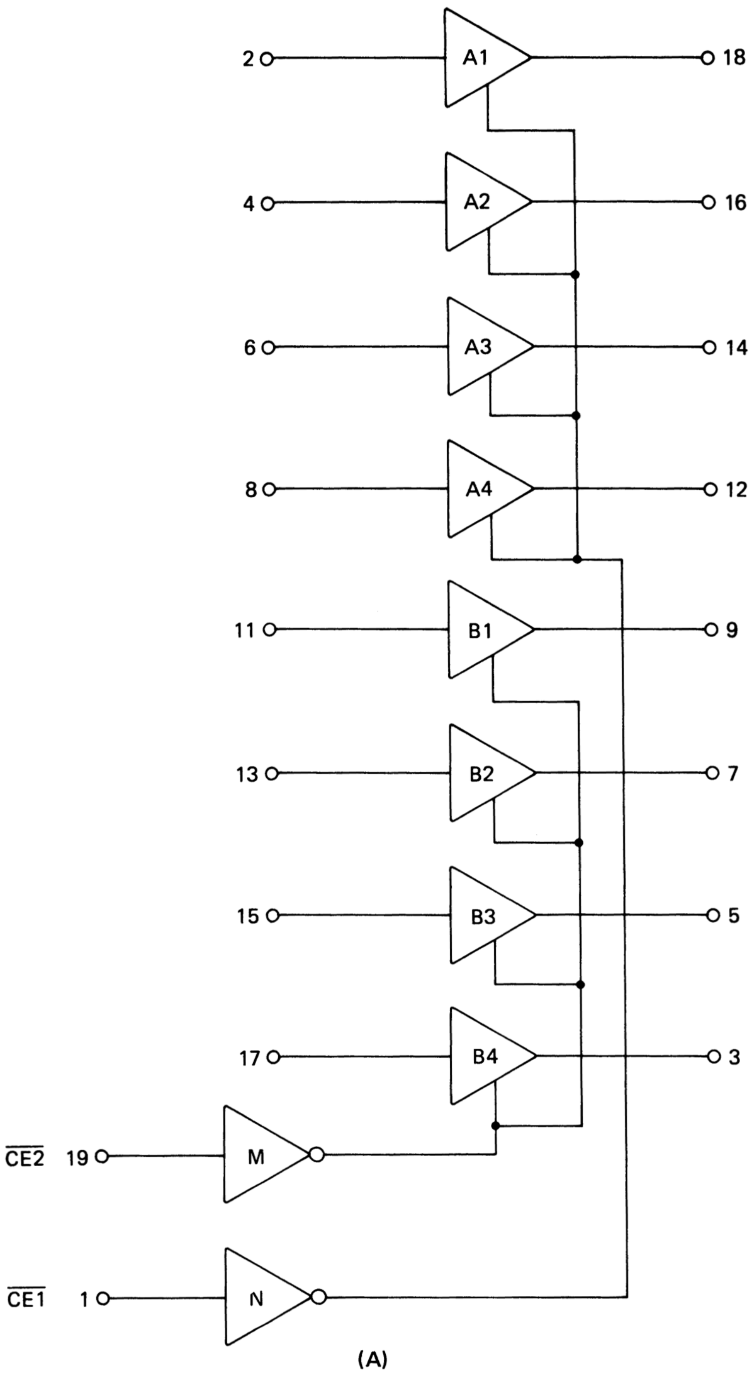
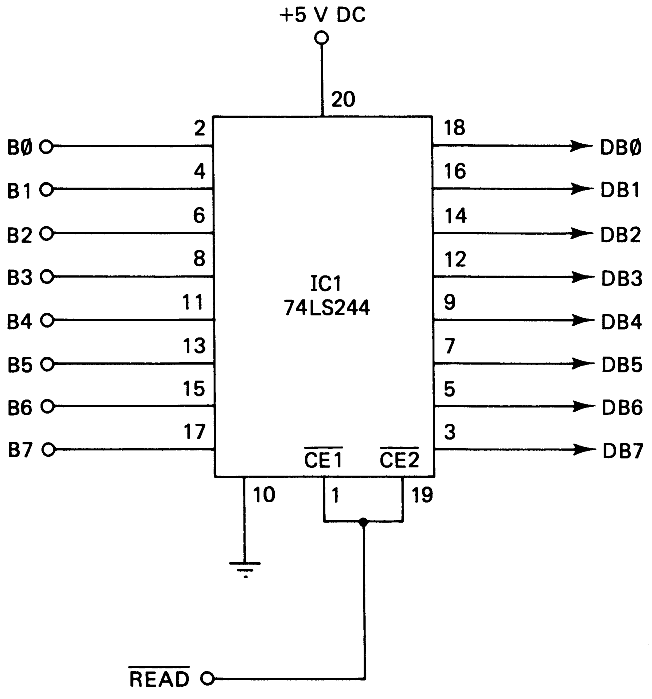


Figure 11-13. A) 74LS244 internal circuit.



(B)

Figure 11-13 (continued). B) 74LS244 as input port

superseded by newer and more powerful microprocessors, some of the support chips still find wide application.

Figure 11-14 shows the internal structure (simplified) for the 8216 and 8226 devices. The principal difference between the 8216 and the 8226 is that the 8216 uses noninverting stages while the 8226 uses inverting stages. Note that the two buffers in each stage are facing in opposite directions with respect to the data bus line (DB0). In other words, the output line of I is connected to the data bus so stage I can be used as an input port line. Similarly, the input of O is connected to the data bus, thereby allowing use of O as an output line. The DI and DO lines are for input and output, respectively.

Control of the 8216 and 8226 devices is through the $\overline{\text{DIEN}}$ and $\overline{\text{CS}}$ inputs. Figure 11-15 shows the truth table that applies to these chips. The chip select line ($\overline{\text{CS}}$) is active-LOW, so we find that the output will be in the high impedance state if $\overline{\text{CS}}$ is made HIGH. The $\overline{\text{CS}}$ line must be LOW for the device to operate. The data direction ($\overline{\text{DIEN}}$) line will connect the input lines (DI) to the data bus (DB) when the $\overline{\text{DIEN}}$ is LOW, and connect the data bus lines to the output lines (DO) when $\overline{\text{DIEN}}$ is HIGH.

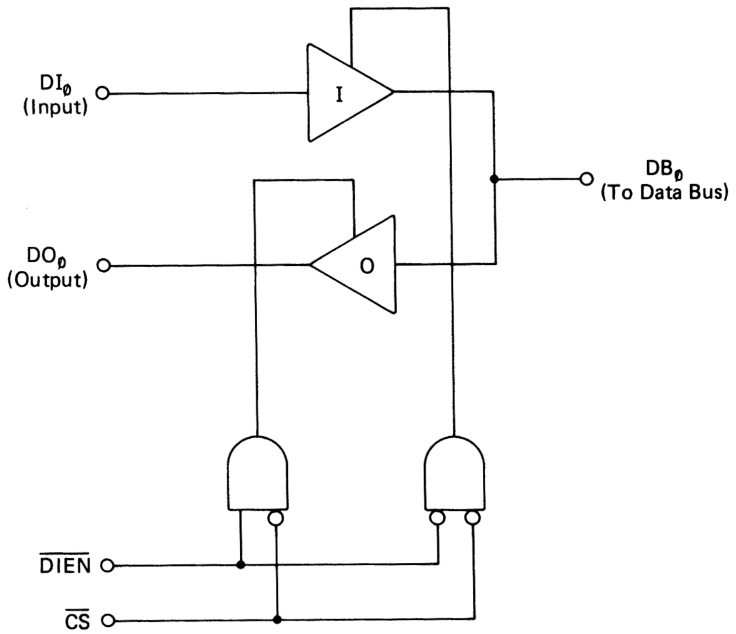


Figure 11-14. One segment from 8216/8226 chips

Figure 11-16 shows two alternate plans for connecting the 8216 and 8226 devices to actual microprocessor circuits. Figure 11-16A shows the basic connections to make these devices work properly, while Figure 11-16B shows a method for using a pair of 8216 devices with a 6502 microprocessor chip. The control signals from the microprocessor chip are specifically designed for use with the 8216/8226 devices.

8216/8226

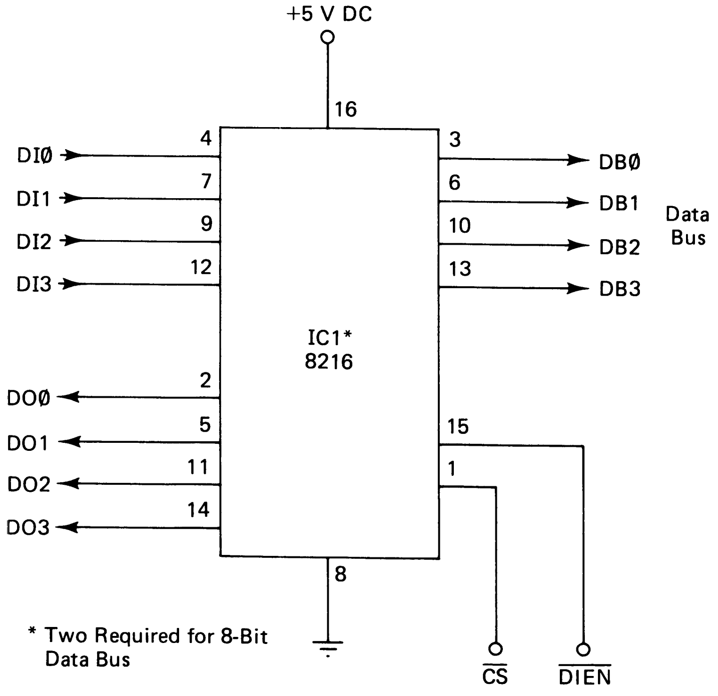
$\overline{\text{CS}}$	$\overline{\text{DIEN}}$	STATE
0	0	DI → DB
0	1	DB → DO
1	X	High-Z
1	X	Output
0 = Low, 1 = High, X = Either		

Figure 11-15. 8216/8226 truth table

INTERFACING KEYBOARDS TO THE MICRO-COMPUTER

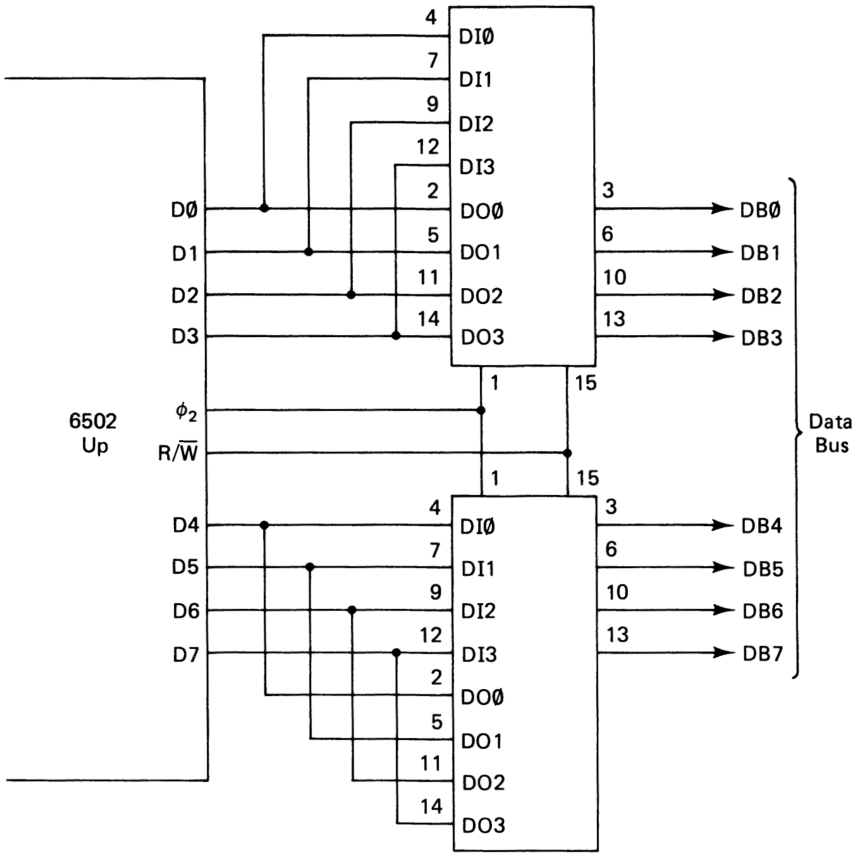
The microcomputer is able to communicate with humans through means of various displays (e.g., video CRT, strip-chart recorder, seven-segment LEDs). The so-called "real world" can communicate with the computer through transducers and data converters. But humans have to communicate with the computer through a device such as a keyboard. The purpose of the keyboard is to allow the human operator to send uniquely encoded binary representations of alphanumeric characters, special symbols, or that denote special functions to the computer. If the computer has been programmed to recognize these special codes, then the human operator can direct the operation of the computer, feed it data, etc.

There are at least three general types of keyboard. First is the simple hexadecimal keypad, which will have 16 keys that are labelled 0 through 9 and A through F. The "hex" keypad will produce either



(A)

Figure 11-16. A) 8216 as input port (4-bit), B) two combined for 8-bit operation



(B)

the 4-bit binary representations of the hex numbers 0000 through 1111 or the ASCII representation (note that the ASCII is a 7-bit code of which the lowest order 4 bits are the same as the binary code for hexadecimal). The second form of keyboard is the full ASCII keyboard that contains all of the alphanumeric characters and outputs unique 7-bit ASCII binary codes representing those characters. The several different forms of this type of keyboard offer 56, 64, or 128 characters (the maximum number allowable with 7-bit codes). The 7-bit ASCII code is ideal for 8-bit microcomputers because the binary word length of the character code is only 1 bit less than the word length of the microprocessor. When the strobe or data valid bit is added to the code bits then a single 8-bit word is totally filled and there are no wasted bits.

The third type of keyboard is the custom or special-purpose keyboard used on electronic instrument panels, for point-of-sale terminals designed to be operated by quickly trained Christmas and summer replacement clerks, etc. The custom keyboard may be merely a series of switches that set some input port bits HIGH or LOW depending upon the situation or it may be a general purpose or hexadecimal keyboard with special keycaps that denote special functions. The computer would be programmed to look for the special symbol and then jump to the program that performs the requisite function when the signal is received.

Figure 11-17 shows the circuit for a typical type of keyboard that is based on a Read Only Memory. Addressing the locations of the memory IC (*IC1*) is by shorting together specific row ("X") and column ("Y") input pins. When the "@" key is pressed, for example, the key switch that denotes "@" is used to short together row "X0" and column "Y8" (see character table in Figure 11-17). This combination uniquely addresses the memory location inside IC1 that contains the binary code that represents the ASCII character "@."

Lines DB0 through DB6 are the data lines for the ASCII code, and DB7 is the strobe line. The strobe line is used to tell the outside world that the data on the other seven lines are valid. Normally, "trash" signals will be on those other lines until a key is pressed and the ASCII code appears. By using the strobe line judiciously, we can create a signal that tells the computer when to believe the DB0–DB6 data. In the case of Figure 11-17, the strobe is a pulse that is created by monostable stable multivibrator (i.e., one-shot) IC2.

The two different types of strobe signal are shown in Figure 11-18. The level type of signal is simply a voltage level that becomes active when the key is closed, and remains active until the key is released. In Figure 11-18A, the signal is active-HIGH so pops HIGH when key closure occurs and drops LOW again when the key is released. The alternate form of strobe signal is the pulse as shown in Figure 11-18B. This signal will snap HIGH only for a brief period (often measured in microseconds) and then go LOW again. By the time the operator releases the key (i.e., after dozens of milliseconds), the computer has input the data and gone on to grander and more wonderful things.

It is important to ensure that the type of strobe signal matches the computer and the software being used. Problems that can make an otherwise normal keyboard appear to malfunction involve the duration of the strobe pulse, software that is looking to find one type of strobe signal but the keyboard supplies the other, and inverted strobe

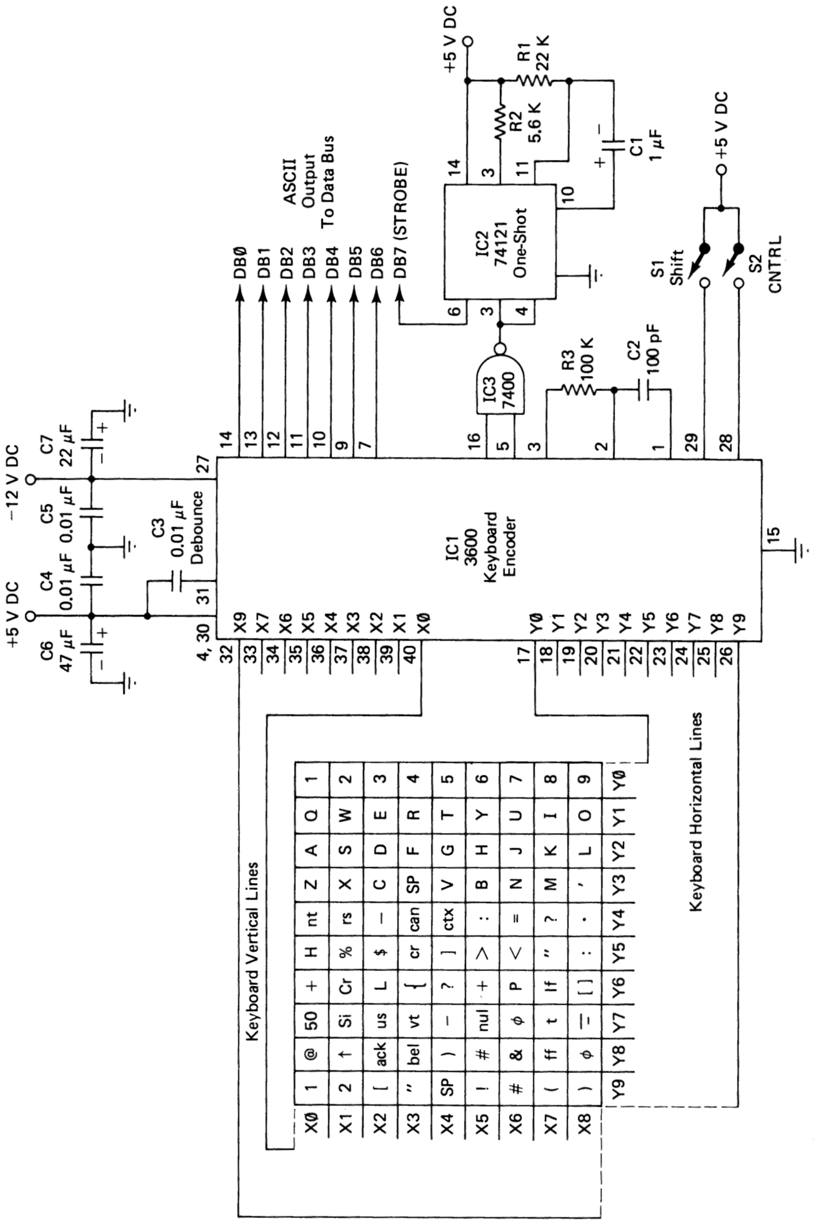


Figure 11-17. Computer ASCII keyboard

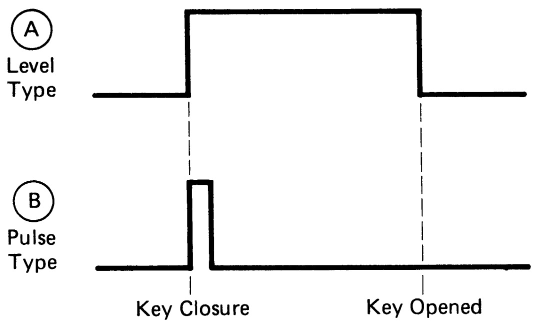


Figure 11-18. Types of strobe signal (A) level, (B) pulse

signals (i.e., the keyboard is active-LOW and the program wants to see active-HIGH so no data-valid strobe signal is received—except when the data is trash). We will discuss possible "fixes" for these problems shortly.

The keyboard is most easily interfaced to the microcomputer that has a spare input port to accommodate it. We can then connect DB0–DB6 to the low order 7 bits of the input port and the strobe signal to the highest order bit of the port. A program is then written to continuously examine that high order bit and branch to the input routine when it sees an active strobe signal. In that case, simple interconnection is all that is needed.

Where there is no available input port, then we may create one using one of the methods shown earlier or some special function I/O port IC device. The I/O port circuitry could then be used to input data from the keyboard directly to the data bus.

Most of the methods for interfacing keyboards will work fast enough that the computer can pick up the valid data each and every time a key is pressed. But at times we will want the computer to come back later and pick up the data (note that "later" could mean 500 ms, but the key would have been released by that time), so that some other program task is not interrupted. In that case, we would want a latched-output keyboard. If the output data on any given keyboard is not latched, then a circuit such as in Figure 11-19 may be used. Here we see the use of another 74100 8-bit data latch. Seven of the latch inputs are used to accommodate the ASCII data lines, and the eighth is not connected. The ASCII strobe signal is used to activate the 74100 strobe lines and will transfer valid ASCII data from the inputs to the outputs of the 74100 so that the computer always sees a valid data signal.

In the case shown in Figure 11-19, the computer must periodically interrogate the input port and take the data each time. Unless there is some reason why the computer must know that the data is new, the

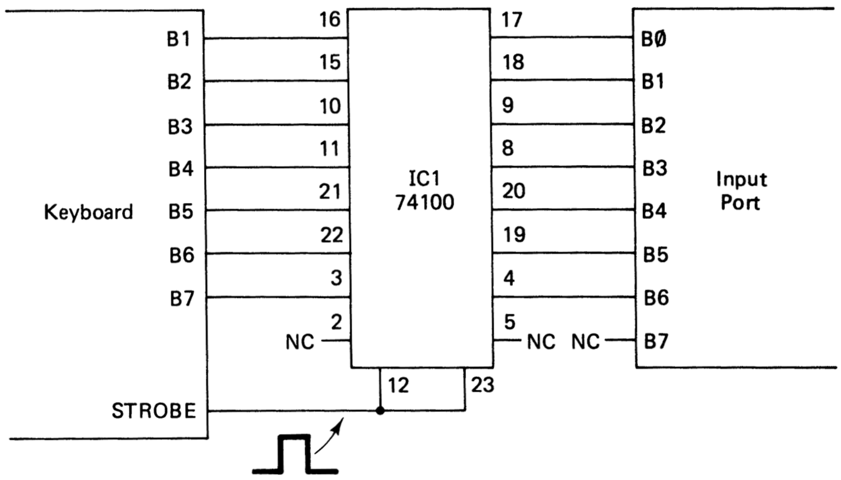


Figure 11-19. Latch circuit for keyboard

strobe is not necessary. We could, however, add a flip-flop that changes state when the strobe signal is received and is then reset when the computer takes the data. In that case, the READ signal generated to activate the input port also could be used to deactivate the strobe FF, provided that the timing could be worked out.

Figure 11-20 shows the fixes for several problems. When the strobe signal is of the wrong polarity, we can interpose an inverter between the strobe output of the keyboard and the strobe line of the computer input port (see Figure 11-20A).

The same basic idea is used when the voltage levels from the keyboard are not compatible with the input level requirements of the microcomputer. It is almost universally true that microcomputers want to see TTL-compatible voltage levels for all signals (i.e., 0-volts and 2.4–5.2 volts for LOW and HIGH, respectively). If the keyboard produces something else, for example, a CMOS logic level, then some form of level translation must be used. The interface device in that case could be a CMOS 4049 or 4050 (depending upon whether inversion is desired) operated from a +5-volt power supply. When the IC is operated from +5 volts DC, then the output lines are TTL-compatible while the input will still accommodate CMOS levels.

Figure 11-20B shows one fix for the situation where the keyboard strobe signal is too short for the microcomputer being used. In many cases, the keyboard used on a microcomputer will seem to malfunction intermittently. The operator will notice that it will not always be picked up by the computer. The problem in that case may well be that the strobe pulse is too short. Microcomputer programs typically loop through several steps that input the data at the port, mask all bits but the strobe, and then test the strobe for either 1 or 0, depending upon whether active-HIGH or active-LOW is desired, and then act accordingly. If the strobe is active, then the program jumps to the input subroutine that will accept the data and stuff it someplace. If, on the other hand, the strobe test shows that it is inactive, then the program branches back to the beginning and inputs the data to test again. It will continue this looping and testing until valid data is received. The problem is that the looping requires a finite period of time to execute—not much time, but still finite. If the strobe pulse comes alive and disappears while the loop program is in another phase than input data, then it will be lost forever. To the operator, it will appear that the computer ignored the keystroke—and much complaining and calling of service technicians will ensue. An example of such a situation is where the computer requires 22 microseconds to execute the loop program, and the keyboard has a 500-nanosecond strobe (they exist!). In that case, we can use the pulse stretcher circuit in Figure 11-20B. The circuit is merely a one-shot, and does not actually stretch anything—it only looks that way to the naive. Actually it uses the strobe pulse from the keyboard as the trigger signal for the one-shot, and then the output of the one-shot becomes the new, longer, and presumedly "stretched," strobe pulse that is sent to the computer. The duration of the pulse is given approximately by $0.7R_1C_1$ and these values can be any normal values under 10 megohms and 10 μF, select values that will make the strobe pulse duration at least long enough that the loop program will catch it, but not so long as to require several loops to outrun it.

Where the low-cost keyboard outputs a level strobe signal, and the computer wants to see a pulse strobe signal, use an arrangement such as in Figure 11-20C. Here we have a 74121 one-shot similar to the one used previously. The difference is that the trigger input is connected to the keyboard strobe line through an RC differentiator (R2 and C2). The purpose of the differentiator is to produce a pulse signal when the level becomes active. Note that, sometimes, one-shot devices will respond to both rising and falling edges, so some sort of diode suppression might be needed in the differentiator output (i.e., trigger input) to eliminate the unwanted version of the signal.

CUSTOM KEYBOARDS, SWITCHES, AND LED DISPLAYS

Custom keyboards may be ordinary keyboards with special keycaps or they may be specially designed sets of switches that tell the computer

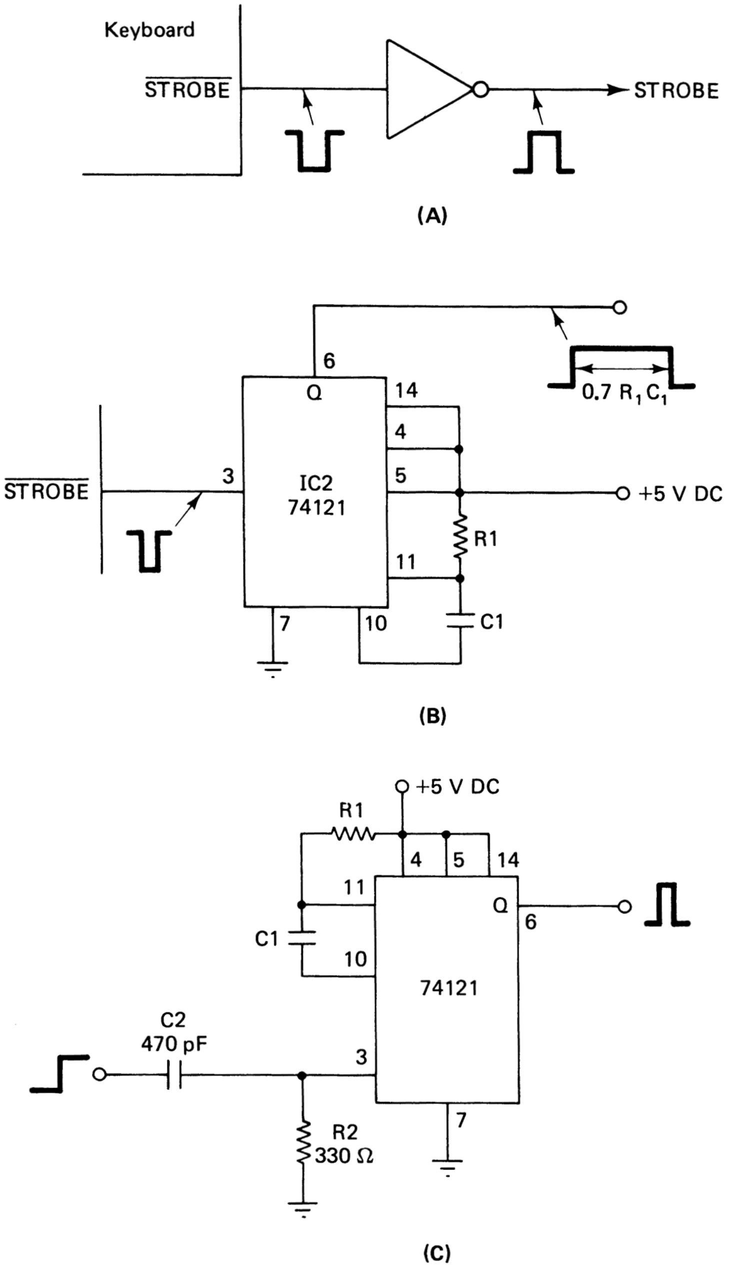


Figure 11-20. A) inverting strobe pulse, B) lengthening strobe pulse, C) circuit for positive-level-to-pulse type strobe signal

to do some neat thing or another. In this section we will consider some of the techniques used to interface and construct these keyboards.

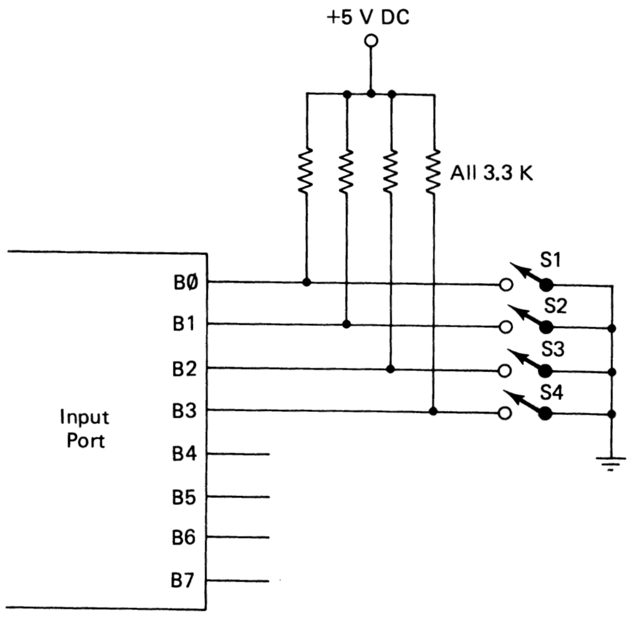
Perhaps the simplest method is that shown in Figure 11-21A. The active element of the keyboard is an input port with switches connected to set each bit either HIGH or LOW. In some cases, especially those where a special-purpose I/O port IC is used, the bits of the port might be ordinarily maintained HIGH by internal pull-up resistors to +5 volts DC, but in most cases we will have to supply the pull-up resistors externally. The resistors are designed to ensure that the open bit of the input port remains HIGH and is not erroneously driven LOW by noise or other factors. The switches will produce a HIGH on the bit line when they are open, and a LOW when they are closed.

Where there is no available input port, then we create one by using a 74LS244 or some similar device to interface the switches to the data bus line. A $\overline{\text{READ1}}$ signal is used to turn on the 74LS244 when the computer wants to read the setting of the switches. The read operation can be either periodic, as in the case of the keyboards, or it may occur just once when the computer is first turned on or the program first begins execution. In this latter case, the computer is asking the keyboard what modes are selected or some similar question. Some designers use this same method to tell the computer which options the customer has purchased. For example, suppose we have a scientific or medical instrument that has eight optional modes that the customer pays for separately from the main instrument. The designer might put a circuit such as in Figure 11-21A on the printed circuit board (the switches are DIP switches) so that the customer engineer or production staff can set them according to which options are purchased. The program to support those options could already be built into the software supplied via ROM to the customer, but only becomes activated when the switch is set to the correct position. Of course, the setting protocol of these switches would be kept confidential lest the customer set them himself, thereby avoiding payment of the license fee.

The example in Figure 11-21B also shows an opto-isolator switch, which is sometimes used to indicate the position of an object. In a popular printer, for example, there is a little metal flange on the print head assembly that will fit into the space between the LED and the phototransistor, thereby blinding the transistor when the print head assembly is at the end of its travel. As long as the transistor sees light, it will be turned on and the state of DB7 will remain LOW. When the print head assembly reaches the limit of travel, however, it will blind the transistor causing it to turn off, and DB7 goes HIGH. The microprocessor used to control the printer carriage will then know to issue the signal that returns the carriage to the left side of the page and issue a line feed signal to advance the paper.

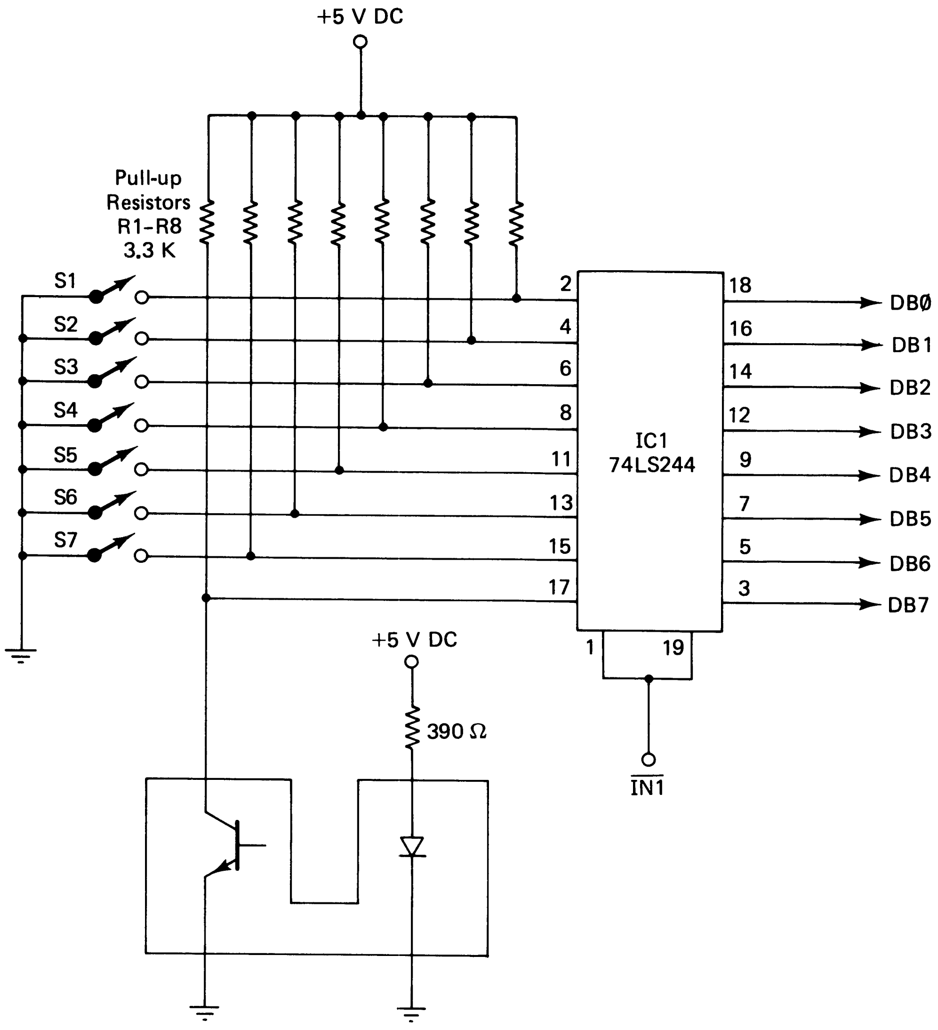
Switches don't make and break in a clean manner; there is almost always some "contact bounce" to contend with. In the case of toggle switches that we set and forget, this "bounce" is not too much of a problem. But in the case of pushbutton switches that are operated regularly, then the contact bounce will produce spurious signals that may erroneously tell the computer to do something besides what the operator intended. The two circuits in Figure 11-22 can be used to "debounce" the pushbutton switches. Figure 11-22A shows the so-called *half-monostable* circuit, which will produce an output pulse with a duration set by R1 and C1 every time the pushbutton switch is operated. The inverter is CMOS type, such as the 4049 or 4050 devices (again, depending upon the desired polarity of the signal). The alternate circuit (see Figure 11-22B) is merely the one-shot circuit used earlier but with a pushbutton switch and pull-up resistor forming the trigger input network. In either case, the output will be a pulse with a duration long enough to allow the bounce signals to die out.

Figure 11-23 shows the methods for interfacing LEDs and LED 7-segment displays to the microcomputer. In both cases, an output port is needed. If none exists, a 74100 or some other device may be



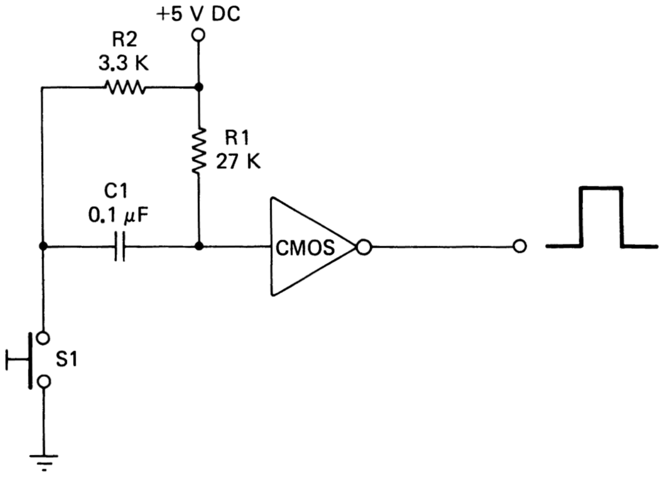
(A)

Figure 11-21. A) interfacing switches for "custom keyboard"

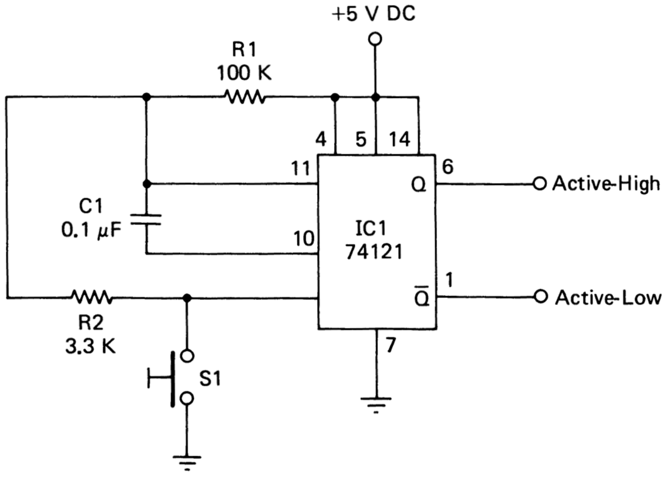


(B)

Figure 11-21 (continued). B) with eight inputs



(A)



(B)

Figure 11-22. Debounced keyboards A) monostable multivibrator type B) half-monostable type

used to form an output port. In the case of Figure 11-23, a single output port is used. Figure 11-23A shows the method for interfacing individual LEDs to the port. Each light-emitting diode is driven by an open-collector TTL inverter. The LED and a current limiting resistor is used to form the collector load for the inverters. The value of the resistor is selected to limit the current to a level compatible with the limits of the LED and the output of the inverter. With the value shown, the current is limited to 15 milliamperes, which is within the capability of most of the available open-collector TTL inverters on the market, and will provide most LEDs with sufficient brightness to be seen in a well-lighted room . . . although not outdoors in direct sunlight.

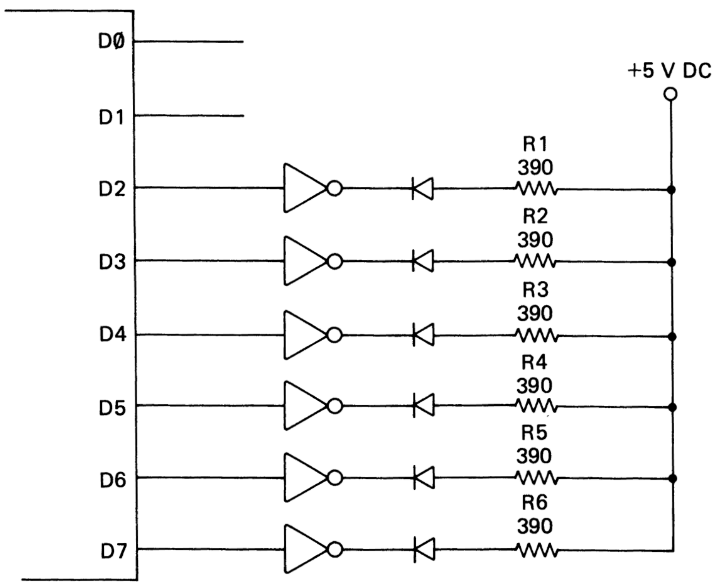
When the input signal of the inverter in Figure 11-23A is HIGH, then the output is LOW, thereby grounding the cathode of the LED. This condition will turn on the LED. Alternatively, when the input of the inverter is LOW, its output will be HIGH so the cathode of the LED will be at the same potential as the anode and no current will flow. Therefore the LED will be off.

Figure 11-23B shows a similar method for interfacing 7-segment LEDs to the microcomputer output port. Here we drive the 7 segments of the LED numerical display device with open-collector TTL inverters in exactly the same manner as with the individual LEDs. This method assumes that the LED numerical display is of the common anode variety with the anode connected to the +5-volt DC power supply.

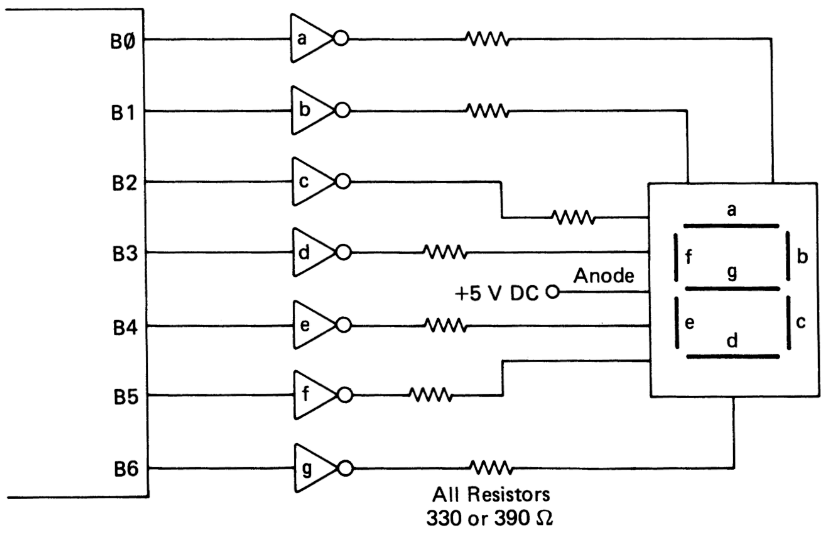
A constraint on this method is that the computer must generate via a software method the 7-segment code. For example, when the number to be displayed is "4," we will want to light up the following segments: f, g, b, and c. These segments are controlled by bits B5, B6, B1, and B2, respectively. Since the segment is turned on when the output port level is HIGH (as in the previous case), we will want to output the binary word 01100110 to turn on the segments that indicate "4." In this case, the decoding of the number "4" into 7-segment code is performed in software, probably using a look-up table.

Figure 11-24 shows a method for interfacing the display through an ordinary TTL BCD-to-7-segment decoder integrated circuit, in this case the 7447 device. The 7447 will accept 4-bit Binary Coded Decimal data at its inputs, decode the data, and turn on the segments of the LED display as needed to properly display that digit. The 7447 outputs are active-LOW, i.e., they drop LOW when a segment is to be turned on and are HIGH at all other times. We therefore would use a common anode 7-segment LED display for this application.

The BCD code applied to the inputs is weighted in the popular 8-4-2-1 method, and according to our connection schema shown in Figure 11-24: B0 = 1, B1 = 2, B2 = 4, and B3 = 8.



(A)



(B)

Figure 11-23. A) Interfacing light-emitting diodes (LED), B) interfacing light-emitting diode seven-segment numerical displays

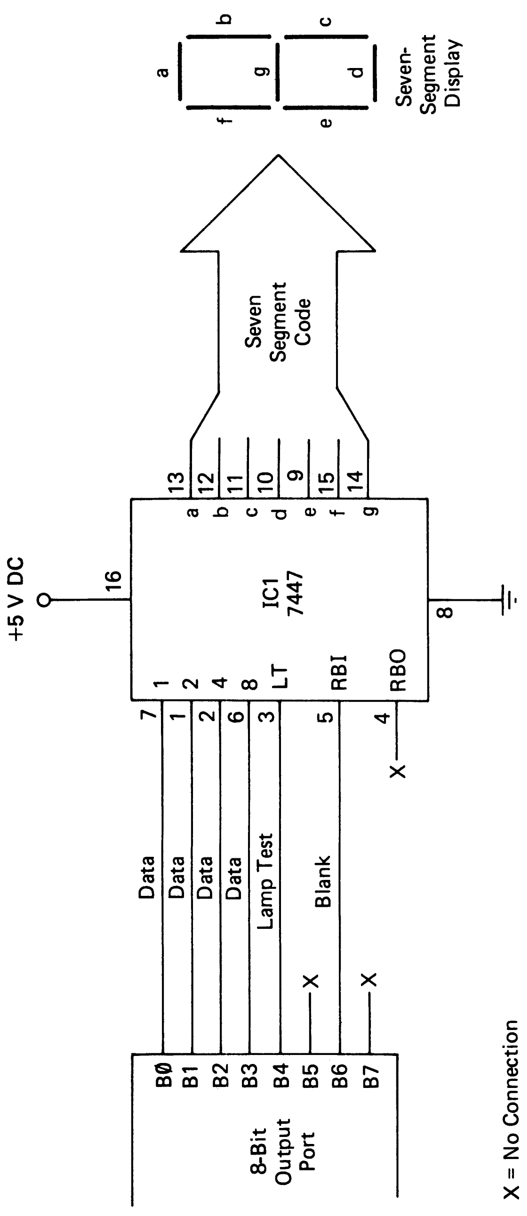


Figure 11-24. Using external decoder for the display

Three control terminals are available on the 7447 device. We have a lamp test (LT, pin 3) that will turn on all 7 segments when it is LOW; at all other times LT is kept HIGH. One function of this terminal is to provide a test of the LED readout to ensure that no burned-out segments exist. The nature of 7-segment readouts allows erroneous readout if one or more segments is burned out or otherwise inoperative. For example, if segment g is defective, an 8 output will read 0! There may be no way for a user to find that defect unless there is a lamp test performed. In some cases, the LT is performed on demand by the user. In that case, a pushbutton switch will ground pin 3 and the user will note whether or not an 8 appears. Of course, all LT terminals of the entire multidigit display can be connected together in one bus to light up all at the same time. In a 6-digit display, grounding the common LT line would produce 888888. The other alternative is to connect the LT line to an output bit of the microcomputer. The program would then display all 8s for a few seconds when the computer or instrument is first turned on, with the idea that the user will observe any defective segments. Be careful when connecting the LT terminals to the output port lest the drive capability of the port bit be exceeded. Most computer output port lines will drive no more than two or three TTL loads, and the LT input represents one such load. In the case where more drive is needed, a noninverting buffer with an appropriate fan-out can be used.

The RBI input is for ripple blanking. If the RBI input is LOW, then the display will turn off if the BCD word applied to the data inputs is zero, i.e., 0000. The purpose is to blank leading zeroes. In other words, without ripple blanking the number "432" displayed on a 6-digit display would read "000432." If we used ripple blanking, however, those three leading zeroes would be extinguished and the display would read "---432." Complementary to the RBI is the ripple blanking output (RBO), which tells the next display that zero blanking is desired. Note that the RBO being grounded will turn off the display, and thus can be used in multiplexing applications.

When using the display in Figure 11-24, a program will load the accumulator with the correct binary coded decimal representation for that digit, and then output it to the port that controls the display. Since microcomputer data words (hence, accumulator registers) tend to be 8 bits or longer, it will be necessary to mask the data to provide zeroes in the high order half-byte of the word. By eliminating the lamp test and blanking features, we can pack the bits in order to make a single 8-bit word contain 2 BCD digits, or, up to 4 BCD digits in a 16-bit word. In the case of the 8-bit accumulator, we could pack the least significant digit (in BCD form) into B0–B3, and the most significant digit into B4–B7 of the 8-bit word. Most common microprocessors have the instructions to automatically accomplish the packing and unpacking of BCD data.

As long as only 1 or 2 digits are required or sufficient output ports are available, the method shown here will be satisfactory. But where output ports must be created, or a large number of digits exist, then we might want to consider multiplexing the displays. In a multiplexed (MUX) display, each digit is turned on in sequence and no 2 digits are on at the same time. If the multiplexed rate is rapid enough, the human eye will blend the on-off transitions and will not notice the flicker. The human eye has a persistence of approximately 1/13 second (i.e., 80 ms), so we will want to switch through the displays at a rate that allows each digit to be turned on before the eye persistence gives it a chance to be recognized. In the case of 6-digit display, therefore, we would want to switch at a rate faster than 80 ms/6, or 12.8 ms. If we take the reciprocal of time, we will find the switching frequency, which in this case would be 1/0.0128s or 78 Hz. We can, therefore, apply an 80 Hz or higher clock and still meet the persistence requirements of the eye. In most cases, however, faster clock rates are used with the attendant smoothing of the display.

So why multiplex? Besides the reduce complexity and chip count of the circuit (hence, improved reliability), there is also the advantage of improved current drain requirements. A typical LED device wants to see 15 mA per segment. If the digit "8" is displayed, with all 7 segments lit, then the current per digit would be 15 mA $\times$ 7, or 105 mA. In the case of our hypothetical 6-digit display, then, we would need 6 $\times$ 105 mA or 630 mA for the display alone! That's more than a half ampere to light display segments . . . and may well be greater than the allowable current budget in many applications (hand-held instruments, such as calculators, need to MUX the display to have a battery life that is even reasonable).

Figure 11-25 shows a method of using a single 7447 device to drive a larger number of 7-segment readouts. The a–g segment lines are bused together so that all a lines, all b lines, etc., are connected into a single line. Therefore, there will be 7 lines feeding the 7 segments of all digits. In the case shown, we would need 21 lines to individually address all 7 segments of all 3 digits. In this arrangement, only 7 lines are used, and the anodes of each digit are connected to the power through transistor switches that are turned on sequentially.

The BCD data is fed to the 7447 through output port 1, while the MUX information is fed to the bases of the control transistors (Q1–Q3) through output port 2. If 4 or less digits are used, then we can conspire to use only one output port, with the BCD data supplied

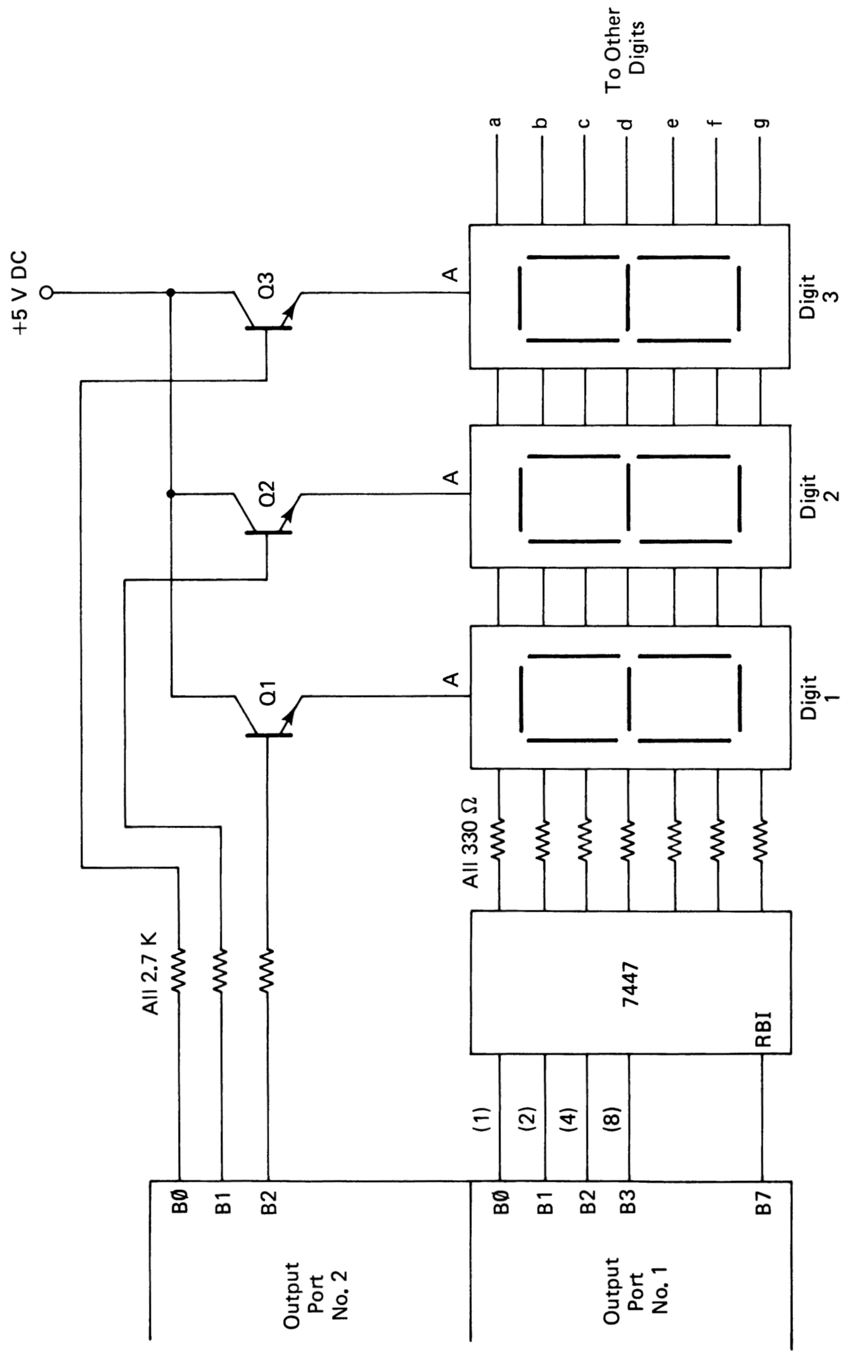


Figure 11-25. Multiplexed display

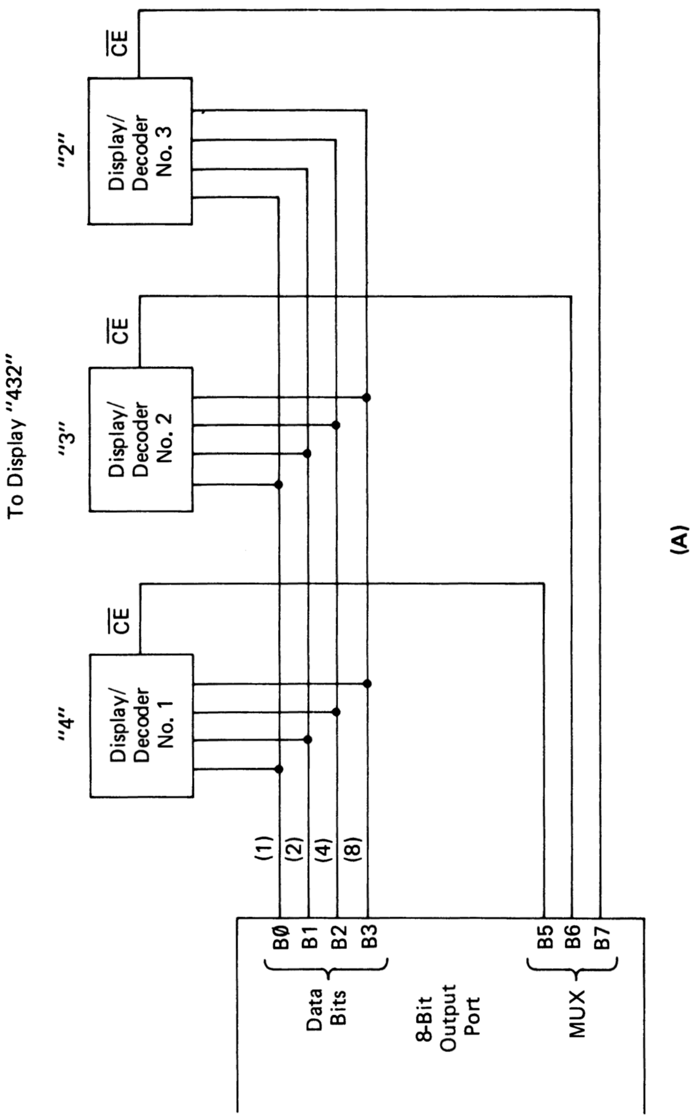
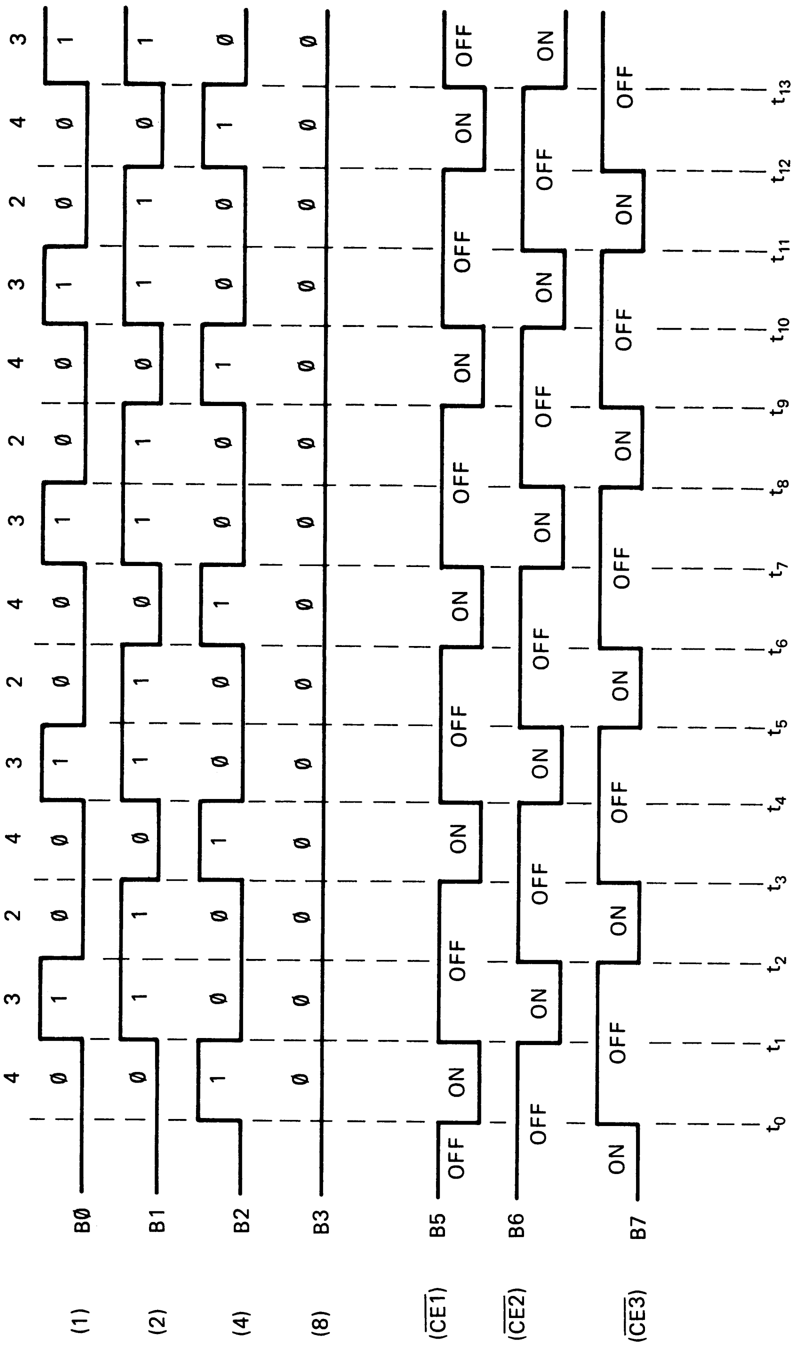


Figure 11-26. Multiplexed display example A) circuit.



(B)

Figure 11-26 (continued). B) timing diagram through B0–B3, and the control bits through B4–B7. Alternatively, we could also add a 7442 BCD-to-1-of-10 decoder to control up to 10 digits, thereby making fuller use of the binary nature of the output port. In that case, the low-order 4 bits (B0–B3) would contain the BCD code, while B4–B7 would contain a BCD word that sequences 0000 through 1001. Let's see what would be needed to make Figure 11-25 display the number "432." We know that the port 2 bits must be HIGH to turn on a digit, so the sequence will be:

Decimal No.	*Port 1**	*Port 2***
4	0100	001
3	0011	010
2	0010	100
4	1000	001
3	0011	010
2	0010	100
4	1000	001
3	0011	010
2	0010	100
.	.	.
.	.	.
.	.	.

* Bits B4–B7 = 0
**Bits B3–B7 = 0

Figure 11-26A shows a method for connecting the display/decoder circuits to a single output port. In the case shown here, the display/decoder might be an old-fashioned combination of 7447 and an LED display or one of the new combination units that contain both the decoder and the 7-segment LED in a single DIP integrated circuit package (e.g., the Hewlett-Packard units). The 4 BCD lines of all displays are connected to a common 4-bit BCD data bus formed from the 4 low-order bits of the output port. The high 3 bits of the port are used as the MUX control signals. The displays are turned on by an active-LOW chip enable ($\overline{\text{CE}}$) line, so the control bits are required to be LOW when the digit is turned on and HIGH at all other times. The timing diagram for the multiplex display is shown in Figure 11-26B. Note that the chip enable lines $\overline{\text{CE1}}$ through $\overline{\text{CE3}}$ are active-LOW, so will each be LOW one-third of the time, in sequence.

12 Interfacing Peripherals to the 6502

A peripheral is a device external to the computer, and which usually (but not always) functions to allow the computer to communicate with the so-called "outside world." Examples of peripherals include the usual assortment of devices such as teletypewriters, printers, keyboards, card readers, tape readers, and CRT video terminals. The peripheral rubric can also be applied to such diverse devices as A/D or D/A converters, electromechanical relays, sensors of one type or another, and lamps or displays. In this chapter, we will consider general methods that permit interfacing the 6502 with a majority of peripheral devices.

PARALLEL PORT METHODS

Occasionally, a device is found which will permit interfacing through a standard 8-bit parallel port. An example is the standard ASCII keyboard. In that case, 7 of the 8 lines are used to carry data, while the eighth line (usually B7) is used as a strobe signal that lets the computer know when new data is available.

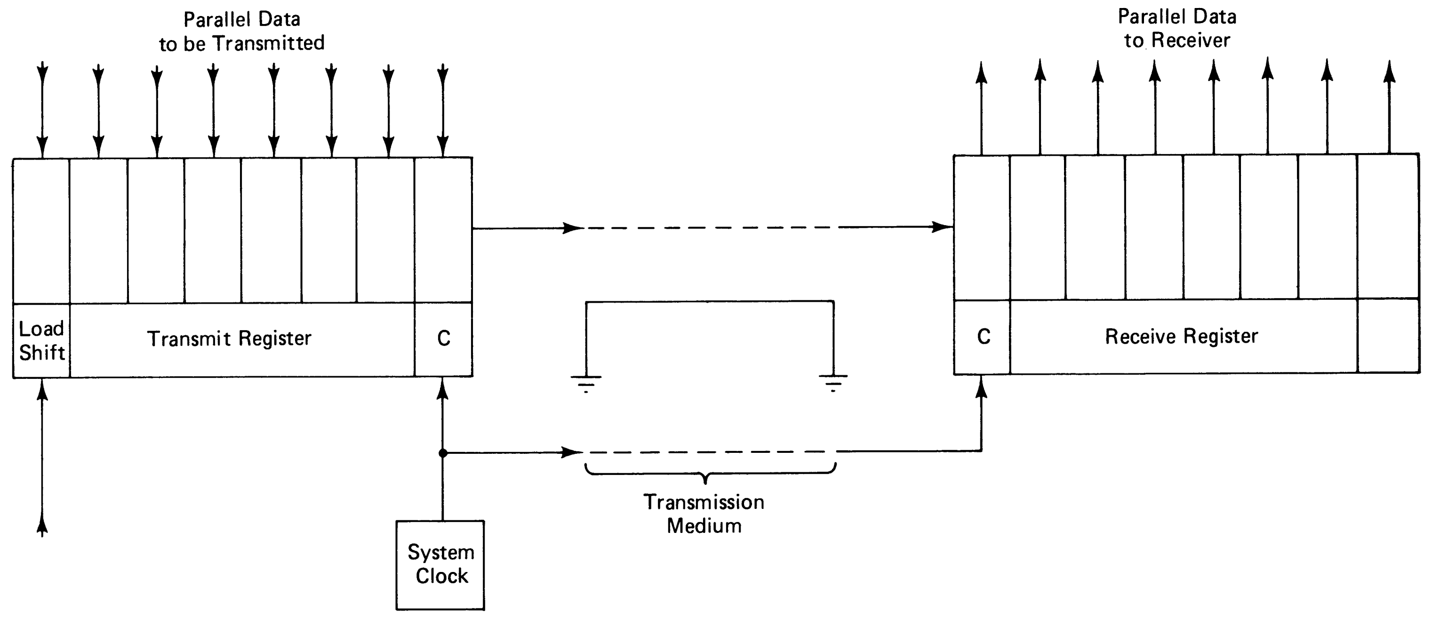
The parallel I/O port is usually very rapid, but also very expensive except within the same computer or over a path of only a few meters outside the computer. For almost all other cases, we will want to use serial data communications methods.

SERIAL DIGITAL DATA COMMUNICATIONS

The interchange of data between machines requires some means of data communication. As mentioned above, parallel communications are probably the fastest method, but can be too expensive for practical application. In parallel communications systems, there will be not less than 1 line for each bit plus a common. For an 8-bit microcomputer, therefore, at least 9 lines are required. In some cases, especially in noisy environments, or where the data rate is very high, it may also be necessary to add additional lines for control or synchronization purposes. Parallel systems are practical over only a few meters distance and are the method generally used in small computer systems for intermachine local connections. But where the distance is increased beyond a few meters or where it becomes necessary to use a transmission medium other than hard wire, e.g., radio or telephone channels, then another method of transmission may be required. For the 8-bit system, for example, we would require not less than 8 separate radio or telephone transmission links between sending and receiving units; that is terribly expensive! The solution is to use only one communications link and then transmit the bits of the data signal serially, i.e., one after another sequentially, rather than simultaneously.

The two forms of serial data communications are diagrammed in Figure 12-1: *synchronous* and *nonsynchronous*. The efficacy of serial communications depends upon the ability of the receiver synchronized with the transmitter. Otherwise, if they are out of sync, the receiver merely sees a series of high and low shifts of the voltage level and cannot make any sense out of the data. The main difference between the synchronous and asynchronous data communications method is in the manner that the receiver stays in step with the transmitter. In the synchronous method, shown in Figure 12-1A, a separate signal is transmitted to initialize the receiver register and let it know that a data word is being transmitted. In some cases, the second transmission medium path will be used to send a constant stream of clock pulses that will allow operation of the receiver register only at certain times. These times correspond to the time of arrival of the data signals. Each bit will be sent simultaneously with a clock pulse. If the incoming signal is LOW when the clock pulse is active, then the receiver knows that a LOW is to be entered into the register, etc.

The problem with the synchronous method is that it requires a second transmission medium path which can be expensive in radio and telephone systems. The solution to this problem is to use an asynchronous transmission system such as shown in Figure 12-1B. In this system, only one transmission channel is required. The synchronization is provided by transmitting some initial start bits that tell the receiver that the following bits are valid data bits. In most systems, the data line will remain HIGH when inert and will signal the intent to transmit a binary word by initially dropping LOW.



(A)

Figure 12-1. A) Synchronous communications

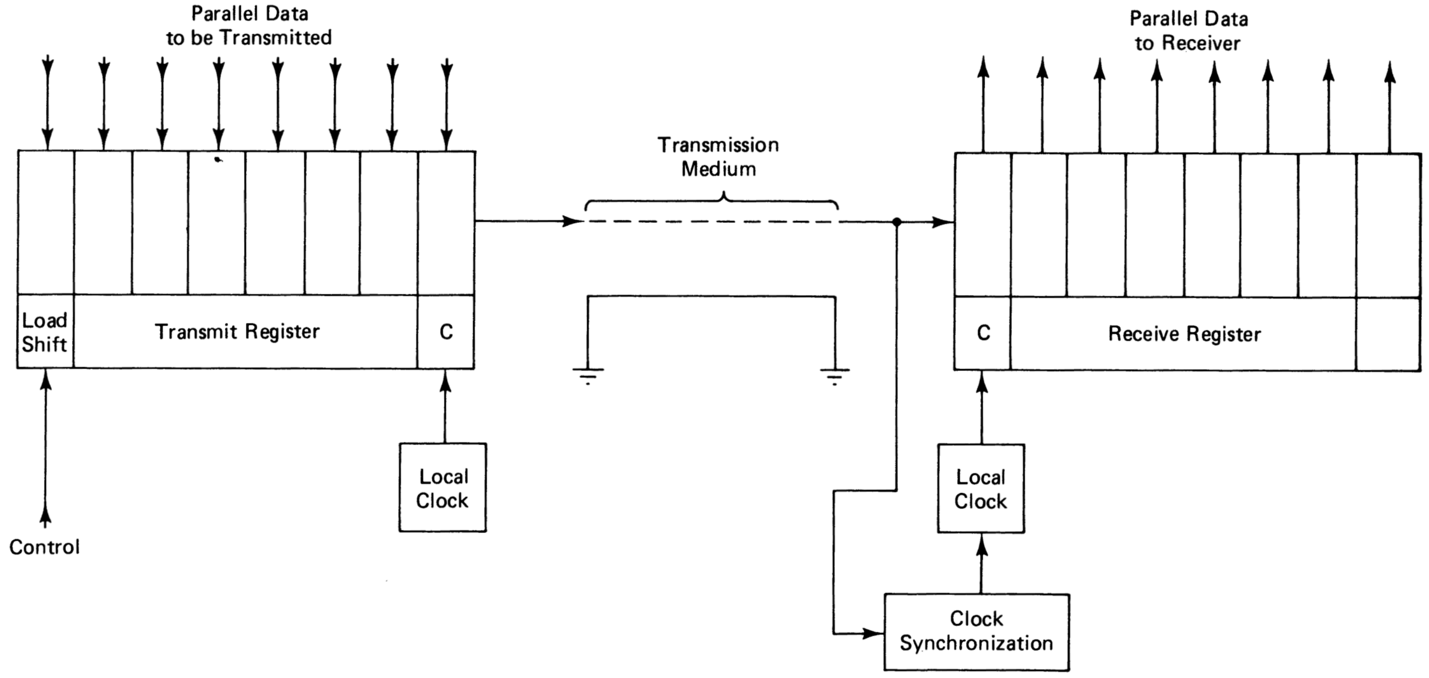


Figure 12-1 (continued). B) asynchronous communications

There are two ways to keep the clock of the receiver synchronous with the transmitter. In one case, an occasional sync signal will be transmitted that keeps the clock on the correct frequency. In most modern systems, however, the receiver clock and the transmitter clock are both kept very accurate, even though locally controlled. Most small computer standards call for the receiver clock frequency to be within either 1 percent or 2 percent of the transmitter clock frequency. As a result, it is typical to find either crystal clocks or RC clocks made with precision low temperature coefficient components.

The design of serial transmission circuits requires the construction of Parallel-In-Serial-Out (PISO) registers for the transmitter, and a Serial-In-Parallel-Out (SIPO) register for the receiver. Each register is designed from arrays of flip-flops, so they can be quite complex.

Fortunately, we can also make use of a large-scale integration (LSI) serial communications integrated circuit called a UART (universal asynchronous receiver/transmitter). Figure 12-2 shows the clock diagram for a popular "standard" UART IC. The transmitter section has two registers: transmitter-hold register and transmitter register. The transmitter hold register is used as a buffer to the outside world, and is a parallel input circuit. The data bit lines from outside of the UART input the data to this register. The output lines of the transmitter hold register go directly to the transmitter register internally, and are not accessible to the outside world. The transmitter register is of the PISO design and is used to actually transmit the data bits. The operation of the transmitter side of the UART is controlled by the transmitter register clock (TRC) input. The frequency of the clock signal applied to the TRC terminal must be 16 times the data transmission rate desired.

The receiver section is a mirror image of the transmitter section. The input is a serial line that feeds a receiver register (a SIPO type). The output register (receiver hold register) is used to buffer the UART receiver section from the outside world. In both cases, the hold registers operate semi-independently of the other registers so can perform certain "handshaking" routines with other circuits to ensure that they are ready to participate in the process.

Like the transmitter, the receiver is controlled by a clock that must operate at a frequency of 16 times the received data rate. The receiver clock (RRC) is separate from the transmitter clock (indeed, the entire receiver and transmitter circuits are separate from each other), so the same UART IC can be used independently at the same time. Most common systems will use the UART in a half-duplex or full-duplex manner so the receiver and transmitter clock lines will be tied together on the same 16X clock line.

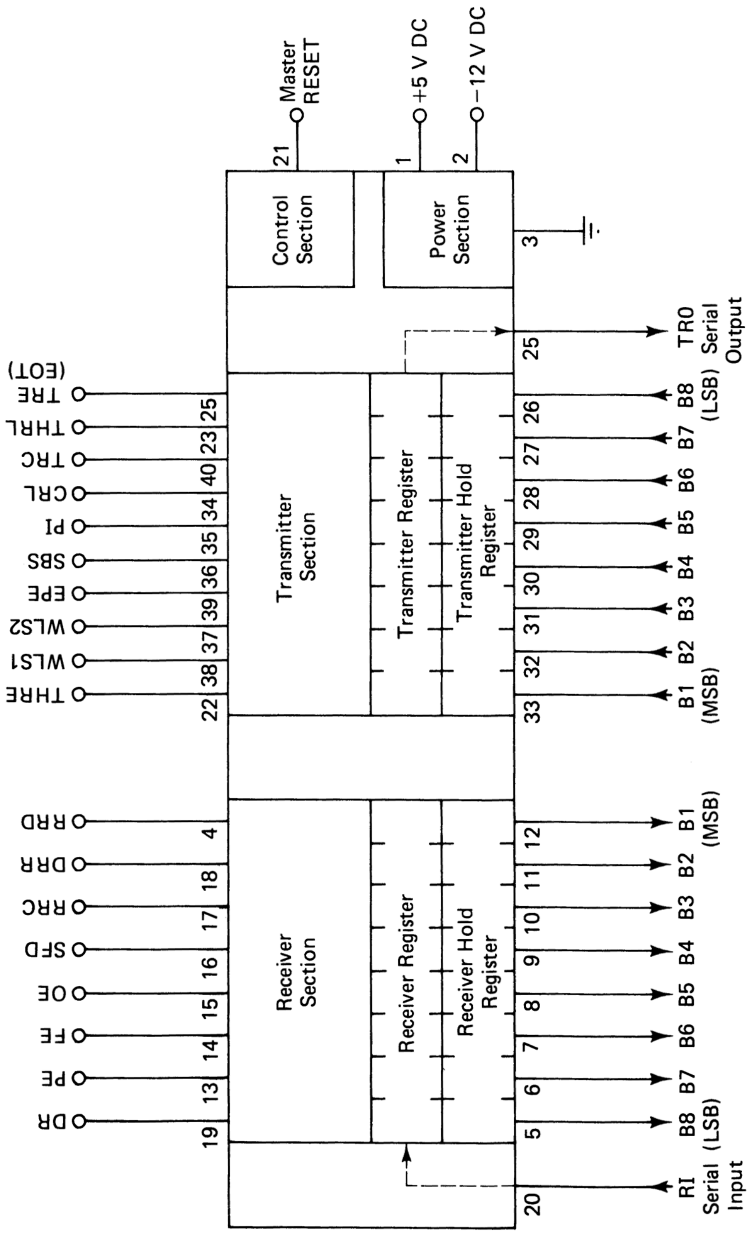


Figure 12-2. Universal asynchronous receiver/transmitter (UART)

The modes of transmission are (1) simplex, (2) half-duplex, and (3) full-duplex. The simplex method transmits data in only one direction. A single UART will be used at the transmit end with the receiver section disabled, while at the receive end another UART is used with an active receive section and a disabled transmit section. In half-duplex transmission, both sections of both UARTs will be used. The half-duplex system is one that has the ability to transmit data in both directions, but only in one direction at a time. The full-duplex method allows the transmission of data in both directions at the same time. Note that, with proper external circuit configuration, most UARTs will support full-duplex communications.

Several control terminals and signals are available on the UART and these aid in operation of the circuit. Some of them, however, may be inactive in any given communications system. The master reset terminal is used to set all registers to zero and return all signals to their inert state. Table 12-1 shows the other signals and control inputs. In a section to follow we will show a typical design for a UART interface with a 6502 microcomputer/microprocessor; therefore we will define only those that are used in that application.

TABLE 12-1

Pin	*Mnemonic*	*Function*
1	V_{CC}	+5 volts DC power supply.
2	V_{EE}	−12 volts DC power supply.
3	GND	Ground.
4	RRD	Receiver Register Disconnect. A high on this pin disconnects (i.e., places at high impedance) the receiver data output pins (5 through 12). A low on this pin connects the receiver data output lines to output pins 5 through 12.
5	RB_8	LSB — Receiver data output lines.
6	RB_7	
7	RB_6	
8	RB_5	
9	RB_4	
10	RB_3	
11	RB_2	
12	RB_1	MSB
13	PE	Parity error. A high on this pin indicates that the parity of the received data does not match the parity programmed at pin 39.

TABLE 12-1 (*Continued*)

Pin	*Mnemonic*	*Function*
14	FE	Framing Error. A high on this line indicates that no valid stop bits were received.
15	OE	Overrun Error. A high on this pin indicates that an overrun condition has occurred, which is defined as not having the DR flag (pin 19) reset before the next character is received by the internal receiver holding register.
16	SFD	Status Flag Disconnect. A high on this pin will disconnect (i.e., set to high impedance) the PE, FE, OE, DR, and THRE status flags. This feature allows the status flags from several UARTs to be bus-connected together.
17	RRC	16 × Receiver Clock. A clock signal is applied to this pin, and should have a frequency that is 16 times the desired baud rate (i.e., for 110 baud standard it is 16 × 110 baud, or 1760 hertz).
18	DRR	Data Receive Reset. Bringing this line low resets the data received (DR, pin 19) flag.
19	DR	Data Received. A high on this pin indicates that the entire character is received, and is in the receiver holding register.
20	RI	Receiver Serial Input. All serial input data bits are applied to this pin. Pin 20 must be forced high when no data are being received.
21	MR	Master Reset. A short pulse (i.e., a strobe pulse) applied to this pin will reset (i.e., force low) both receiver and transmitter registers, as well as the FE, OE, PE, and DRR flags. It also sets the TRO, THRE, and TRE flags (i.e., makes them high).
22	THRE	Transmitter Holding Register Empty. A high on this pin means that the data in the transmitter input buffer has been transferred to the transmitter register, and allows a new character to be loaded.
23	THRL	Transmitter Holding Register Load. A low applied to this pin enters the word applied to TB1 through TB8 (pins 26 through 33, respectively) into the transmitter holding register (THR). A

TABLE 12-1 (*Continued*)

Pin	*Mnemonic*	*Function*
		positive-going level applied to this pin transfers the contents of the THR into the transmit register (TR), unless the TR is currently sending the previous word. When the transmission is finished the THR → TR transfer will take place automatically even if the pin 25 level transition is completed.
24	TRE	Transmit Register Empty. Remains high unless a transmission is taking place, in which case the TRE pin drops low.
25	TRO	Transmitter (Serial) Output. All data and control bits in the transmit register are output on this line. The TRO terminal stays high when no transmission is taking place, so the beginning of a transmission is always indicated by the first negative-going transition of the TRO terminal.
26	TB_8	LSB
27	TB_7	
28	TB_6	
29	TB_5	Transmitter input word.
30	TB_4	
31	TB_3	
32	TB_2	
33	TB_1	MSB
34	CRL	Control Register Load. Can be either wired permanently high, or be strobed with a positive-going pulse. It loads the programmed instructions (i.e., WLS1, WLS2, EPE, PI, and SBS) into the internal control register. Hard wiring of this terminal is preferred if these parameters never change, while switch or program control is preferred if the parameters do occasionally change.
35	PI	Parity Inhibit. A high on this pin disables parity generator/verification functions, and forces PE (pin 13) to a low logic condition.
36	SBS	Stop Bit(s) Select. Programs the number of stop bits that are added to the data word output. A high on SBS causes the UART to send 2 stop bits if the word length format is 6, 7, or 8 bits, and

TABLE 12-1 (*Continued*)

Pin	*Mnemonic*	*Function*
		1.5 stop bits if the 5-bit teletypewriter format is selected (on pins 37-38). A low on SBS causes the UART to generate only 1 stop bit.
37 38	WLS_1 WLS_2	Word Length Select. Selects character length, exclusive of parity bits, according to the rules given in the chart below:
39	EPE	Even Parity Enable. A high applied to this line selects even parity, while a low applied to this line selects odd parity.
40	TRC	16 × Transmit Clock. Apply a clock signal with a frequency that is equal to 16 times the desired baud rate. If the transmitter and receiver sections operate at the same speed (usually the case), then strap together TRC and RRC terminals so that the same clock serves both sections.

Word Length	WLS_1	WLS_2
5 bits	low	low
6 bits	high	low
7 bits	low	high
8 bits	high	high

Data Received (DR). A HIGH on this terminal indicates that the data have been received and are ready for the outside world to accept.

Overrun Error (OE). A HIGH on this terminal tells the world that the data reset (DR) flag has not been reset prior to the next character coming into the internal receive hold register.

Parity Error (PE). Parity error signal indicates that the parity (odd or even) of the received data does not agree with the condition of the parity bit transmitted with that data. A lack of such match indicates a problem in the transmission path.

Framing Error (FE). A HIGH on this line indicates that no valid stop bits were received.

B1–B8 Receiver. Eight-bit parallel output from receiver (tri-state).

B1–B8 Transmitter. Eight-bit parallel input to transmitter.

Transmitter Hold Register Empty (THRE). A HIGH on this pin indicates that the data in the transmitter hold register has been transferred to the transmitter register and that a new character may be loaded from the outside world into the transmitter hold register.

Data Receive Reset (DRR). Dropping this line LOW causes reset of the data received (DR) flag, pin 19.

Receiver Register Disconnect (RRD). A HIGH applied to this pin disconnects (i.e., causes to go tri-state) the B1–B8 receiver data output lines.

Transmitter Hold Register Load (THRL). A LOW applied to this pin causes the data applied to the B1–B8 transmitter input lines to be loaded into the transmitter hold register. A positive-going transition on THRL will cause the data in the transmitter hold register to be transferred to the transmitter register, unless a data word is being transmitted at the same time. In that case, the new word will be transmitted automatically as soon as the previous word is completely transmitted.

Receiver (serial) Input (RI). Data input to the receiver section.

Transmitter Register (serial) Output (TRO). Serial data output from the transmitter section of the UART.

World Length Select (WLS1 and WLS2). Sets the word length of the UART data word to 5, 6, 7, or 8 bits according to the protocol given in Table 12-1.

Even Parity Enable (EPE). A HIGH applied to this line selects even parity for the transmitted word, and causes the receiver to look for even parity in the received data word. A LOW applied to this line selects odd parity.

Stop Bit Select (SBS). Selects the number of stop bits to be added to the end of the data word. A LOW on SBS causes the UART to generate only 1 stop bit regardless of the data word length selected by WLS1/2. If SBS is HIGH, however, the UART will generate 2 stop bits for word lengths of 6, 7, or 8 bits and 1.5 stop bits if a word length of 5 bits is selected by WLS1/2.

Parity Inhibit (PI). Disables the parity function of both receiver and transmitter and forces PE LOW if PI is HIGH.

Control Register Load (CRL). A HIGH on this terminal causes the control signals (WLS1/2, EPE, PI, and SBS) to be transferred into the control register inside of the UART. This terminal can be treated in one of three ways: strobe, hardwired, or switch controlled. The strobed method uses a system pulse to make the transfer and is used if the parameters either change frequently or are under program control. If the parameters never change, then it can be hardwired HIGH. But if changes are made occasionally, the control lines and CRL can be switch controlled.

The UART chip is particularly useful because it can be programmed externally for several different bit lengths, baud rates, parity (odd-even, receiver verification/transmitter generation), parity inhibit, and stop bit length (1, 1.5, or 2 stop bits). The UART also provides six different status flags: transmission completed, buffer register transfer completed, received data available, parity error, framing error, and overrun error.

The clock speed is 320 kHz (maximum) for the A and B versions, 480 kHz for the AO3/BO3 versions, 640 kHz for the AO4/BO4 versions, and to 800 kHz for the AO5/BO5 series. The receiver output lines are tri-state logic, so will float at a high impedance to both ground and the +5-volt line when inactive. The use of tri-state output allows the device to be connected directly to the data bus of a computer or other digital instrument.

The transmitter section uses an 8-bit parallel input register that will accept data to be sent serially. It will convert the 8-bit data word received in the input register to serial format that includes the 8-bit word (also formattable to 5, 6, or 7 bits), start bit, parity bit, and stop bits.

The receiver can be viewed as simply the mirror image of the transmitter. It receives a serial input word containing start bits, data, parity, and stop bits. This serial stream of data is checked for validity by comparison with parity and for the existence of the stop bits.

The UART data format (serial) is shown in Figure 12-3. The transmitter output pin will remain HIGH unless data are being transmitted. Start bit B0 is always HIGH-to-LOW transition, which tells the system that a new data word is about to be sent. Bits B1 through B8 are the data bits loaded into the transmitter on the sending end of the system. All 8 bits of the maximum word length format are shown in the figure, even though truncated word lengths of 5, 6, or 7 bits are also allowable. The stop bit length can be programmed for 1, 1.5, or 2 bits, according to the needs of the system designer.

The number of data bits, the parity, and the number of stop bits are programmed onto the device using HIGH and LOW levels applied to certain pins designated for that purpose. For example, the WLS1

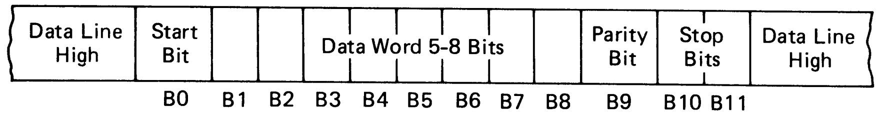


Figure 12-3. Serial communications data word

and WLS2 pins are used as word length select pins, and will set the data word length according to the following code system:

Word Length	*WLS1*	*WLS2*
5 bits	0	0
6 bits	1	0
7 bits	0	1
8 bits	1	1

Similarly, a 2-bit stop code is selected by connecting SBS HIGH, but only when the data word is 6, 7, or 8 bits. If the data word is set to 5 bits length, which is used on Baudot teletypewriters, then the 1.5-bit stop code is used. If SBS is LOW, then the stop code is 1 bit in length. The parity is set by the EPE pin, and will be coded odd for a LOW and even for a HIGH.

The clock on a UART system must be stable, so we cannot generally use RC timer-based clocks and expect proper performance, especially at high baud rates. The frequency of the clock must be 16 times the baud rate. If we want to transmit data of 300 baud, for example, the oscillator frequency must be 300 × 16, or 4800 Hz. While this frequency is well within the range normally competent RC oscillators can produce, it is recommended that a crystal oscillator be used to ensure the stability and accuracy of the clock. An attractive alternative is the CMOS 4060 device, which contains an internal crystal or RC oscillator and a chain of binary divider stages.

The transmitter circuit is shown in Figure 12-4A. Note that only the 8-bit input data, clock, and serial output are required to make this circuit operational. The TRE, THRE, and THRL signals are status flags, and are optional although they will probably be used in most practical cases. They convey information about the status of the information transfer, and are sometimes needed in the software used to control the UART. A careful review of the meaning of each flag is necessary for designers who wish to use the UART.

The basic receiver circuit for the UART is shown in Figure 12-4B. We have a similar simplicity in the receiver section (one of the principal attractions held by LSI devices to equipment and instrument designers). Only the clock, serial in/out, and 8-bit parallel output lines are needed. Again, however, certain signals are available that will make some applications either easier or possible; these are the DR, OE, FE, and PE flags. Table 12-1 gives the meanings of these signals, and those of the transmitter section.

Notice in the receiver section that we use an inverter from the data received terminal to reset the DRR terminal. This signal tells the

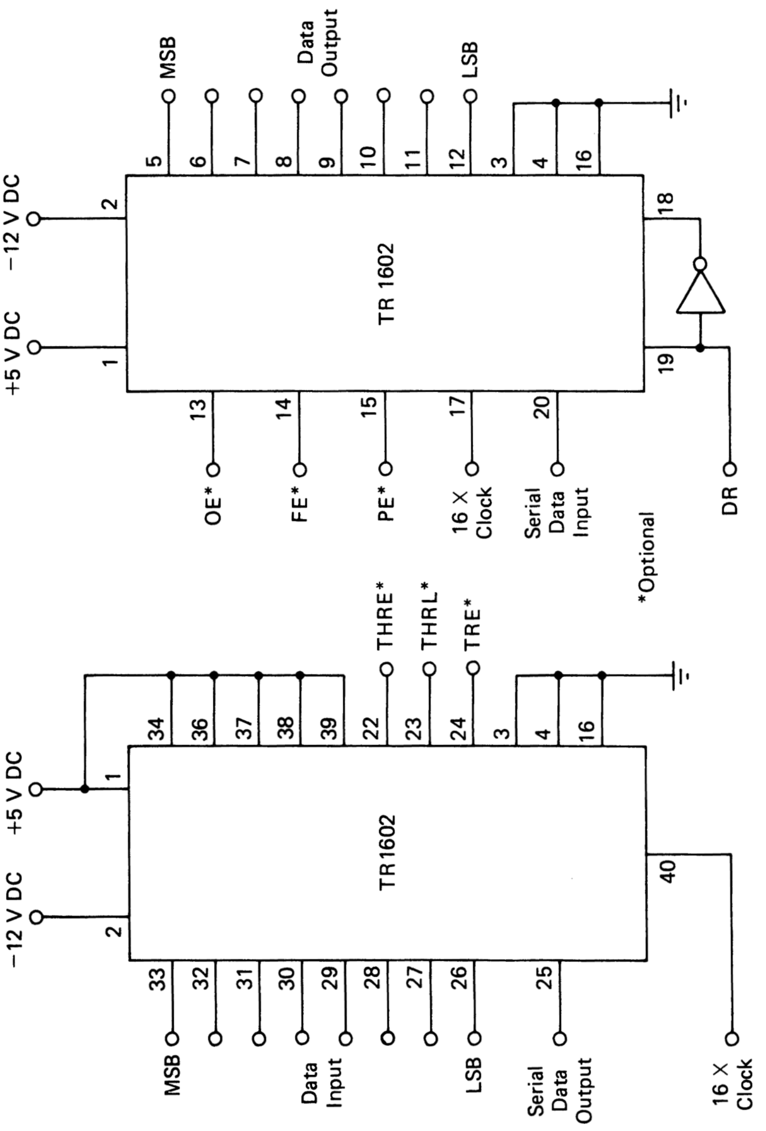


Figure 12-4. (A) TR1602UART-transmitter connections (B) receiver connections for the TR1602

UART to get ready for the next character and can be used to signal a distant transmitter that the UART is ready to receive another transmission.

One thing about the UART that appeals to many designers is that the two sections (receiver and transmitter) can be used either independently or in a common system. In a simplex communications channel (one direction only), a transmitter-wired UART is used on the transmitter end, while a receiver-wired UART is used on the receiving end. In a half-duplex system (bidirectional communication, but only one direction at a time), both sections are used at each end, and the status flags can be used in a handshaking system to coordinate matters. Full-duplex operation is possible, but requires either a second channel (especially in radio links) or a second set of audio tones in hard-wired telephone line systems. Not all telephone lines, however, are amenable to full duplex operation, especially over long distance lines.

In dedicated instrument applications, the programming pins will probably be hard-wired in the proper codes, but in many case switches are used to allow the user to program as needed. You can also connect the UART control pins to an I/O port to permit programming of the UART under software control of the computer.

An example of a "standard" UART configured for use with the 6502 microprocessor is shown in Figure 12-5. Because of the nature of the UART, we can use it directly as an I/O port, and memory-map it to the 6502 without the need for any external circuiting except device select signals.

The transmitter input lines are high impedance, so can be connected directly to the 6502 data bus. Similarly, the receiver output lines are tri-state, so will float at high impedance (neither HIGH nor LOW) until the receiver is turned on. Therefore, we can connect both receiver and transmitter directly to the data bus (DB0–DB7). Also connected to the data bus are the DR, OE, PE, FE, and THRE.

The UART is programmed by the CRL, PI, SBS, EPE, WLS1, and WLS2 pins being made HIGH or LOW. The protocols governing these control pins were given earlier. In Figure 12-5, the control pins are set by switches. Each input is tied HIGH through 3.3K pull-up resistors. If the corresponding switch is open, therefore, that input is HIGH, but if the switch is closed, the input is shorted to ground (and therefore is LOW).

The control input scheme of Figure 12-5 assumes that we need variable control over the UART programming. The use of DIP switches on the UART printed wiring board permits us to set these factors almost at will. If such a capability is not needed, however, we can also hard-wire the inputs HIGH or LOW as needed.

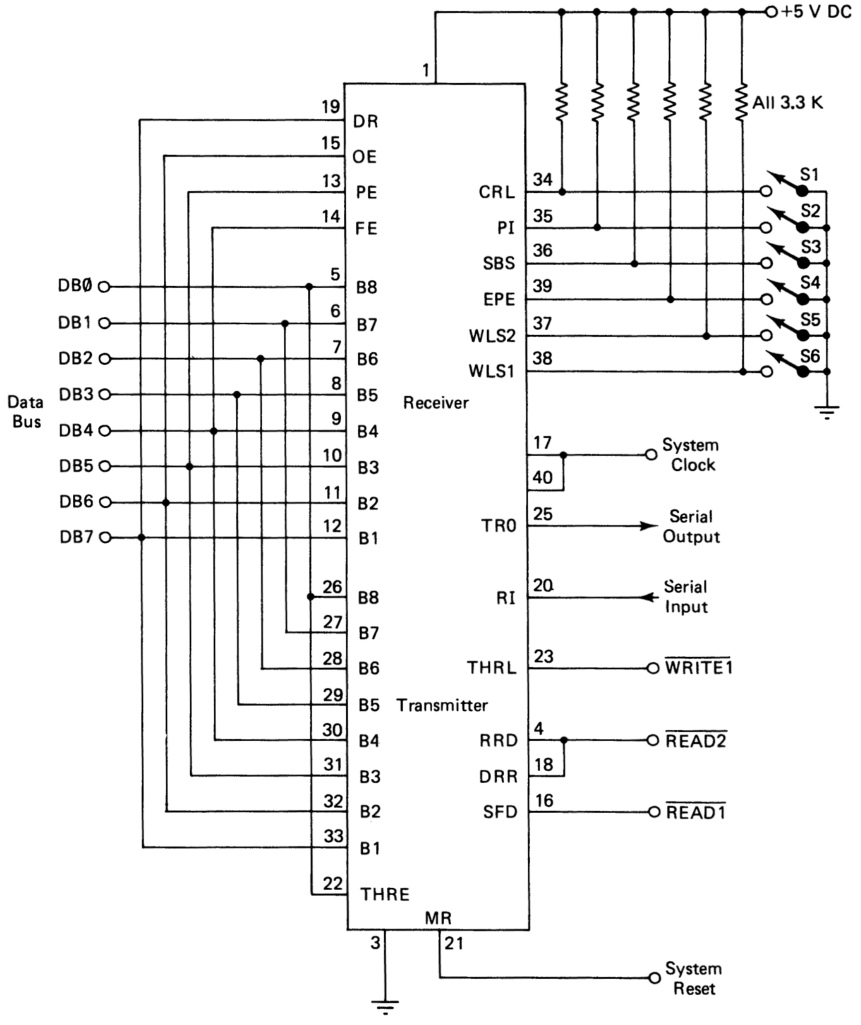


Figure 12-5. Practical UART interfacing

A variation on the theme is to connect the control input lines to a latched output port. We could, for example, use 6 bits of a 74100 device to contain our HIGH/LOW states. The inputs of the 74100 would be connected to the data bus, while the inputs are connected to the UART control lines. If we memory-map the 74100, and provide suitable device select circuitry, then simple write operations will allow us to set the UART parameters under program control.

RS-232 Interfacing

The Electronic Industries Association (EIA) standard RS-232 pertains to a standardized serial data transmission scheme. The idea is to use the same connector (i.e., the DB-25 family), wired in the same manner all the time, and to use the same voltage levels. Supposedly, one could connect any two devices that provide RS-232 I/O without any problem; it usually works.

Modems, CRT terminals, printers (i.e., Model 43 Teletypewriter), and other devices will be fitted with RS-232 connectors. Some computers provide RS-232 I/O; this feature can be added by using a set of Motorola ICs called RS-232 drivers/receivers. An RS-232 driver IC accepts TTL outputs from a computer or other device, and produces RS-232 voltage levels at its output. The RS-232 receiver does just the opposite. It takes RS-232 levels from the communications/interface and produces TTL outputs.

Unfortunately, the RS-232 is a very old standard, and it predates even the TTL standard. That is why it uses such odd voltage levels for logical-1 and logical-0.

Besides voltage levels, the standard also fixes the load impedances and the output impedances of the drivers.

There are actually two RS-232 standards—the older RS-232B and the current RS-232C (see Figure 12-6). In the older version, RS-232B, logical-1 is any potential in the -5- to -25-volt range, and logical-0 is anything between $+5$ and $+25$ volts. The voltages in the range -3 to $+3$ are a transition state, while $+3$ to $+5$ and -3 to -5 are undefined.

The speedier RS-232C standard narrows the limits to ± 15 volts. In addition, the standard fixes the load resistance to the range 3000 to 7000 ohms, and the driver output impedance is low. The driver must provide a slow rate of 30 volts/microsecond. The Motorola MC1488 driver and MC1489 receiver ICs meet these specifications.

The standard wiring for the 25-pin DBM-25 connector used in RS-232 ports is shown in Table 12-2.

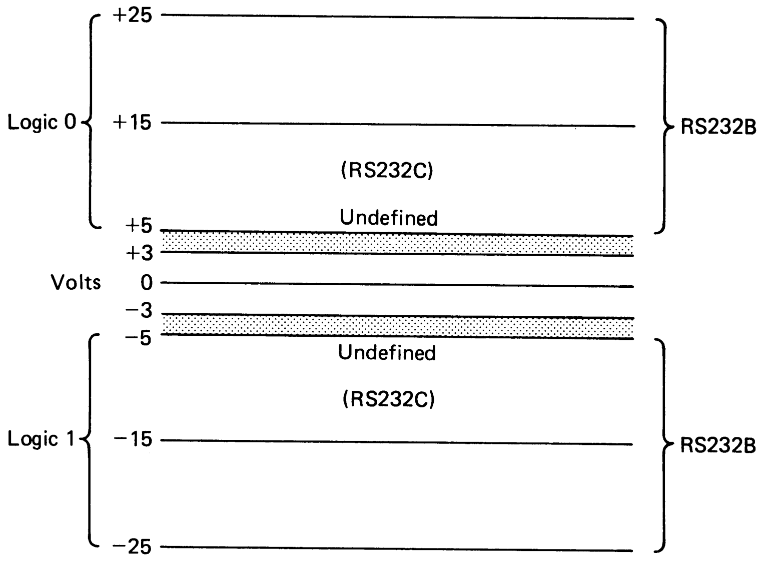


Figure 12-6. RS-232B/C serial communications logic levels

Current Loop Ports

The current loop port was designed specifically for use with teletypewriters, but it has been adopted over the years to a variety of communications problems in digital instruments. The original 60 (and later 20) milliampere current loop systems were intended for Baudot Teletype machines, and were used to energize the solenoids in the printer. But the same idea has also been adopted for use with a variety of printers other than teletypes and is also found in certain other instruments that must communicate with computers. The 60 mA version of the current loop is obsolete but is included here because it is often necessary to design into older existing systems.

Figure 12-7A shows the most basic circuit for a 60-mA machine. An external 130-volt DC power supply is needed. The current loop circuit consists of the DC supply, resistor R2, the teletypewriter machine, and *c-e* path of transistor Q1.

Diode D1 is used as a spike suppressor. The solenoid coils will produce a spike-like pulse (i.e., high amplitude, short duration) every time the current flow in one of the coils is interrupted. Diode D1 is connected to suppress these spikes, and is used mainly to protect transistor Q1.

Transistor Q1 can be any high-voltage power transistor that is capable of handling a 60-mA collector current. Q1 acts as a switch to turn the loop on and off.

If a HIGH appears on the LSB of the selected output port, then Q1 is forward-biased. Its *c-e* path conducts current, closing the loop. When the LSB of the output port is LOW, then Q1 is reverse-biased. Under this condition, its *c-e* path is turned off, so the loop is open.

It is best to adjust resistor R2 to obtain a loop current to 60 mA. Place a HIGH on the LSB of the selected port, and press one of the

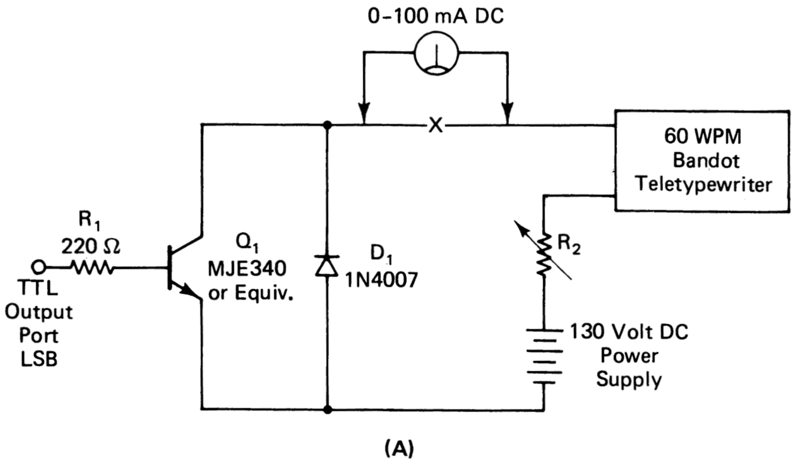
TABLE 12-2

EIA RS-232 Pin-outs for Standard DB-25 connecta

Pin	*RS-232 Name*	*Function*
1	AA	Chassis ground
2	BA	Data from terminal
3	BB	Data received from modem
4	CA	Request to send
5	CB	Clear to send
6	CC	Data set ready
7	AB	Signal ground
8	CF	Carrier detection
9	undef	
10	undef	
11	undef	
12	undef	
13	undef	
14	undef	
15	DB	Transmitted bit clock, internal
16	undef	
17	DD	Received bit clock
18	undef	
19	undef	
20	CD	Data terminal ready
21	undef	
22	CE	Ring indicator
23	undef	
24	DA	Transmitted bit clock, external
25	undef	

teletypewriter keys. A millimeter placed at the point indicated in Figure 12-7A will show the current. Adjust the resistor (R2) for a flow of 60 mA.

It is probably best if all high-voltage circuits are isolated from your computer's output. A fault in transistor Q1 could otherwise cause damage to the output port circuits. An appropriate circuit for this is shown in Figure 12-7B. The secret is to use an optoisolator device. On



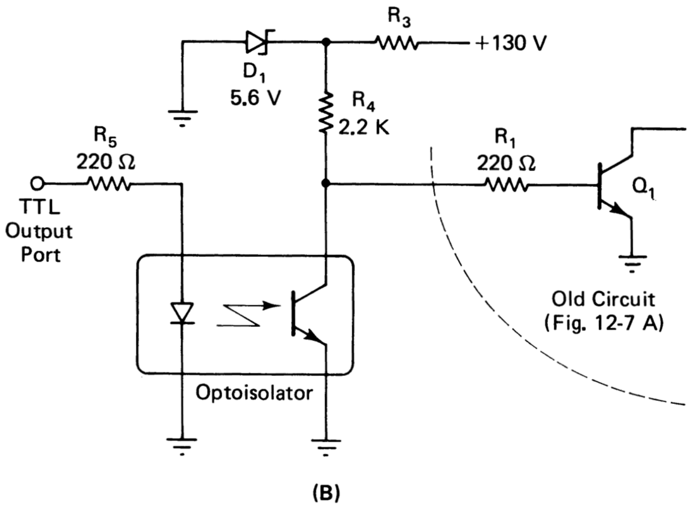


Figure 12-7. A) Simple circuit to interface old-style Baudot teletypewriters. Adjust R2 for 60 mA in the loop, B) Circuit above modified to isolate the teletypewriter from the computer output circuitry.

the computer side of the device is an LED, while on the teletypewriter side is an optotransistor. The transistor will be turned off unless the LED is turned on. The collector of the optoisolator transistor is connected to the point in the previous circuit that connected to the computer. This collector is also connected to a 5.6-volt DC power supply that is derived from the +130-volt DC power supply used in the current loop. On the computer side, the LED is connected through a current-limiting resistor (R5) to the LSB of the selected port.

When the LSB of the output port is HIGH, then the LED is turned on. This turns on the transistor in the optoisolator, shorting out the bias to the current loop transistor. This action turns off the loop. Similarly, the LOW on the LSB of the port turns off the transistor, so Q1 is turned on, closing the loop. The action in this circuit is inverted, so it is necessary to complement the 6502 accumulator before outputting data. Alternatively, one other transistor inverter could be used, between the isolator and Q1, to invert the output of the isolator.

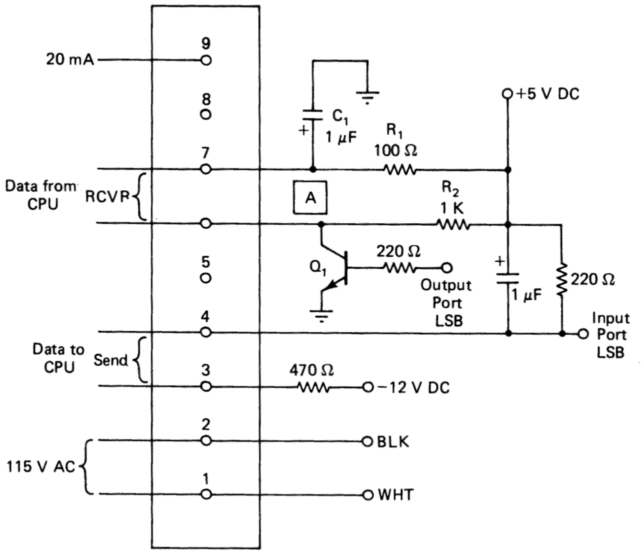
Figure 12-8A shows a circuit that is used to interface the model 33 to an output port. Looking from the front panel, there is a terminal strip on the right-rear side of the Model 33. This terminal strip, shown schematically in Figure 12-8B, contains the send/receive connections for the teletypewriter.

The receive side of the machine (terminals 6 and 7) contains the loop, so that the solenoids can be pulled in. The send side is merely a set of contact closures. In my own experience, this circuit has produced some problems. If the loop is turned on after the microcomputer is loaded and ready to work, a random pulse seems to change a few (important) bits in a few memory locations. The problem is partially relieved by using +5-volt and −12-volt power supplies that are completely divorced from the computer power supply. But I like the approach shown in Figures 12-8B and 12-8C. We would use R1, R2, and C1 (from Figure 12-8A), but replace Q1 with the transistor from the optoisolator (connect the collector point A). The LED is connected, again through a current-limiting resistor, to the LSB of the selected output port.

We can use the −12-volt supply to drive the LED, or the +5-volt supply, in which case the polarity is reversed. The isolator transistor (Q1) drives an inverter stage (Q2). When the LED is turned on, Q2 is turned off, so the LSB of the selected input port is HIGH. But if the LED is off, then Q2 is turned on, dropping the LSB of the input port to zero.

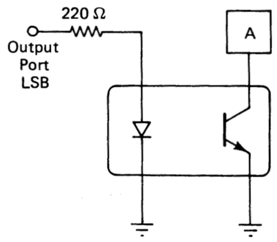
Serial Interfacing

The three basic forms of serial data line are: TTL, 20 mA Current loop, and RS-232. In the TTL version, one bit of a TTL-compatible I/O port



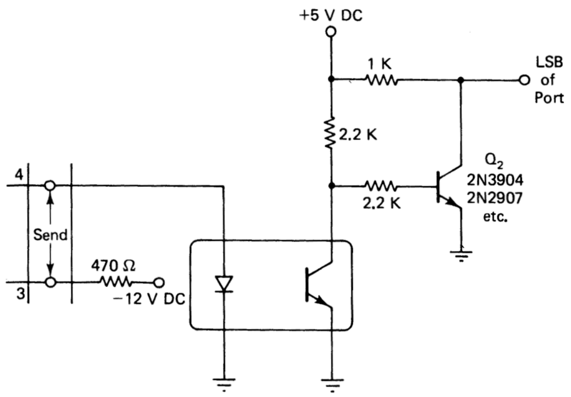
20 mA
9
8
7
Data from CPU
RCVR
A
5
4
Data to CPU
Send
3
2
1
115 V AC
C_1
1 μF
R_1
100 Ω
R_2
1 K
+5 V DC
220 Ω
Q_1
Output Port LSB
1 μF
220 Ω
Input Port LSB
470 Ω
−12 V DC
BLK
WHT

(A)



220 Ω
Output Port LSB
A

(B)



+5 V DC
1 K
LSB of Port
2.2 K
Q_2
2N3904
2N2907
etc.
2.2 K
4
Send
3
470 Ω
−12 V DC

(C)

is used as the serial output or, alternatively, the serial output of a TTL-compatible UART is used. The 20-mA and RS-232C have already been defined. With so many standard systems, we often face an interfacing chore of trying to make two units with dissimilar serial ports play together. We might, for example, want to connect a Model-33 Teletype^R machine to the RS-232 output of a 6502-based computer. Or alternatively, we might want to interface a TTL port with either (or both) 20 mA or RS-232. This activity seems especially common among hobbyists, universities, and other small users who obtain surplus equipment or otherwise find themselves forced by budget constraints to mix equipment.

The job of interfacing these various kinds of serial ports is basically one of level translation. The TTL device, for example, produces 0 to 0.8 volt when LOW, and +2.4 volts or more when HIGH. Furthermore, the TTL port may be capable of sinking only 1.8 mA on LOW (i.e., will drive only one TTL input) or it may be buffered sufficiently to sink 50 to 100 mA. The RS-232 port, on the other hand, uses ±5 to ±15 volts (+12 is very common) for HIGH/LOW levels. The 20 mA loop presents still another translation problem, i.e., conversion of a current level to either TTL or RS-232C voltage levels. In this section, we will discuss some of the more popular conversion schemes. First, however, we will take a closer look at the 20 mA current loop.

Figure 12-9 shows detail of a typical 20 mA current loop serial data communications system. This system will only work when the loop is closed. When both keyboard (transmitter) and printer (receiver) are part of the same machine, we must either wire them in full duplex (i.e., receiver and transmitter separate) or provide a send-receive switch (S1) across the keyboard terminals. Also, if we want a local capability, then we must either connect the TTY into a system with another machine, or, provide a local switch that shorts the output. The switch is shown in dotted line form in Figure 12-9. If this switch is closed, then the teletypewriter keyboard will "talk" to its own printer even though the equipment is disconnected from the network.

The transmitter is a keyboard, and can be modelled as a switch that closes when the operator presses a key (the actual operation is more complex than this simple model). The receiver (printer) can be

Figure 12-8. A) Circuit to connect computer output port to the Model 33 teletypewriter. Terminal block shown is found under the top cover of the Mdl. 33, on the right rear when viewed from the front of the keyboard. Use separate +5 volt DC power supplies for best results, B) Modification of the standard circuit to allow isolation of the computer from the teletypewriter, C) different circuit to accomplish the same job

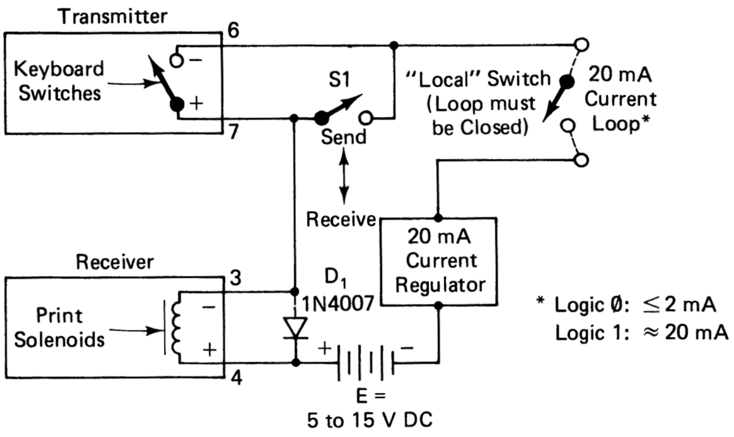


Figure 12-9. Teletypewriter circuit

modelled as a solenoid coil in series with the line. This fact can be important in digital circuits because the de-energized solenoid coil will produce a voltage spike if the instant current flow in the coil ceases. The diode, D1, is used to suppress that spike. The diode is normally reverse biased when current flows, so the diode is effectively out of the circuit. When current flow ceases, however, a reverse polarity counterelectromotive force (CEMF) is generated that forward biases the diode. Thus, the peak of the CEMF spike, which would otherwise be hundreds of volts, reduces to 0.7 volt (the junction potential of D1).

The power supply shown in Figure 12-9 usually produces a voltage of 5 to 15 volts, and will include a 20-milliampere current regulator. In some machines, the regulator is a solid-state circuit, but in most it is a resistor.

Figure 12-10 shows a simple circuit that will convert 20-mA current loop signals to either TTL (most common) or CMOS logic levels. This circuit provides a high degree of isolation between the TTL and 20-mA sides. Without isolation, noise or simple dynamic load changes caused by the 20-mA machine would affect the computer. Total isolation requires that the 20-mA circuit have its own separate power supply.

The 4N35 optoisolator contains a light emitting diode (LED) positioned such that its light falls on the base of a phototransistor (Q1). The entire assembly is inside a DIP integrated circuit package. Diode D1 protects the LED by suppressing spikes on the line. If the 20-mA loop is well regulated, then resistor R is not needed. Its function is to limit the current to a safe value to protect the LED, and its value is set by the actual current in the loop.

The TTL side of the circuit consists of the optoisolator phototransistor (Q1), resistor R1, and an inverter. If the circuit is TTL, then R1 will be 470 ohms or so, and the supply voltage is +5 VDC. Of course, IC2 will be a TTL inverter. Let's consider how the circuit works.

Recall that the HIGH current in the loop is 20 mA, while the LOW current is 0 to 2 mA. During the HIGH periods, therefore, the LED is on, and during LOW periods it is off. Transistor Q1 is controlled by the LED when the LED is on; during LOW periods it is off. When the LED is lighted, indicating a HIGH on the loop, transistor Q1 will be on. This condition results in the collector-emitter resistance of Q1 being very low. The input of the inverter will be LOW under that condition, so its output will be HIGH.

Similarly, a LOW on the loop turns off the LED, so the phototransistor is also off. Under this condition the Q1 collector-emitter resistance is high, so the input of the inverter will see a HIGH. Thus, the output of the inverter will be LOW. In both cases, the output of IC2 is the same logical value as the current loop. The output of the inverter is connected to one bit of an I/O port or to a TTL-compatible serial data input.

If the circuit of Figure 12-10 interfaces to a CMOS circuit or computer port, then it will be necessary to use a different power supply

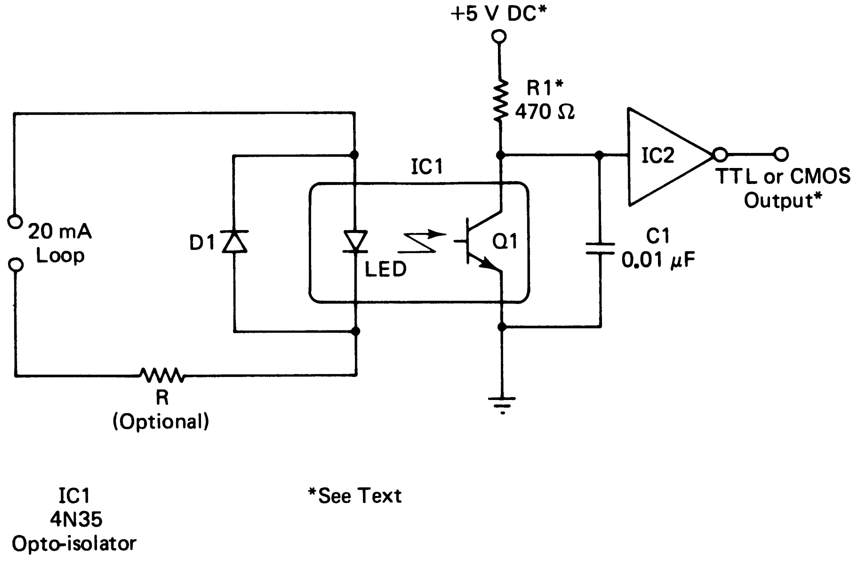


Figure 12-10. Isolated 20-milliampere loop interfacing (input to computer)

voltage, and must change R1 proportionally. The idea is to keep the current flowing in Q1 at a safe value.

At some of the higher data rates, the anti-noise capacitor C1 may tend to dampen the signal too much. The solution to that problem is to remove C1 or, at least, reduce its value (perhaps to 0.001μF).

The opposite interface circuit is shown in Figure 12-11; this circuit converts TTL or CMOS data to 20-mA current loop signals. The inverter will be an open-collector TTL type or a Type-B CMOS device. In the case of the CMOS device, no series resistor is needed, provided that the supply voltage is +5 VDC.

When the data input line is LOW, the output of IC2 is HIGH. Under this condition, the potential is the same at both ends of the LED, so no current flows. The LED is turned off, so Q1 is also off. The loop current will be zero, indicating a LOW bit.

When the data input line is HIGH, the opposite occurs. The output of IC2 is LOW, so the cathode of the LED is effectively grounded. The LED is therefore turned on, as is transistor Q1. The collector-emitter resistance of Q1 is low at this point, so current flows in the loop. In this circuit, you may have noticed, transistor Q1 acts as an electronic switch; it will always be either fully on or fully off.

Diode D1 is used to suppress noise spikes on the 20 mA loop. The 1N4007 selected for this application has a PIV rating of 1000 volts, and a forward current rating of 1 ampere.

Figure 12-12 shows RS-232C versions of Figures 12-10 and 12-11. Recall from earlier in the chapter that RS-232C is a standard which uses −5 to −15 volts for logical-1 (HIGH) and +5 to +15 volts for

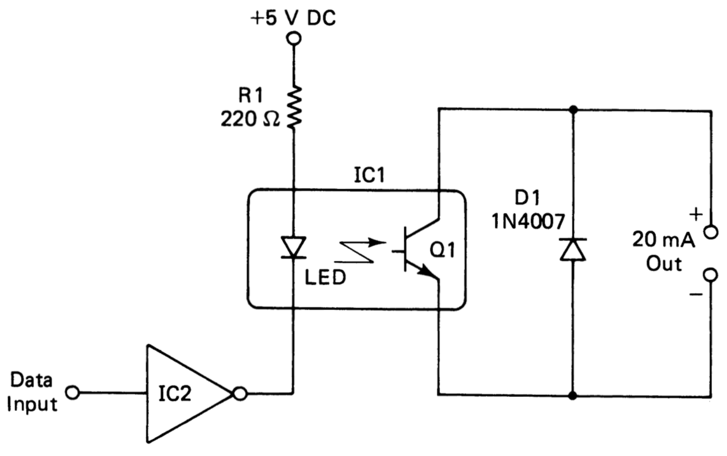


Figure 12-11. Isolated 20-milliampere loop interfacing (output from computer)

logical-0 (LOW). When the current loop is LOW (less than 2mA), the LED is turned off, so Q1 (Figure 12-12A) will be off, or if the switch analogy is used, Q1 is open. Under this condition, point A will be at a potential of +12 VDC, so the output level, according to the RS-232C convention, is LOW. This level, of course, matches the logic level on the 20 mA current loop. When the loop is HIGH, i.e., when 20 mA is flowing, the LED is on, as is Q1. Under this condition, the collector-emitter resistance of Q1 is very low so the voltage at point A will be a little less than −12 VDC. This voltage is the RS-232C HIGH level.

The opposite number is shown in Figure 12-12B: This circuit converts RS-232C serial data signals to 20 mA current loop signals. When a LOW is applied to the RS-232C input line, the output of the second inverter will also be LOW (+12 VDC). This condition means that both ends of the LED are at the same potential; the LED is off. Because the LED is off, the transistor Q1 is also off; current on the loop is zero. When a HIGH is applied to the RS-232C input, the output of the second inverter will be at −12 VDC, so current will flow and turn on the LED. Since the LED is turned on, the collector-emitter resistance of Q1 is low, so the "switch" is effectively turned on and current flows in the loop. This condition is the HIGH for a 20-mA current loop.

A TTL-to-RS-232C interface circuit is shown in Figure 12-13. This circuit is based on the popular 741 operation amplifier, which connects as a voltage comparator. The rules of voltage comparator operation are:

1. When V1 = V_{REF}, the output is zero.
2. When V1 is greater than V_{REF}, then the output will be at the maximum negative output voltage.
3. When V1 is less than V_{REF}, then the output will be at the maximum positive output voltage.

Since V_{REF} is +1.4 volts, a V1 TTL HIGH logic level (i.e., over +2.4 volts), will satisfy condition 2, so the output of the operational amplifier will be high negative (approximately −8 to −10 VDC). This voltage level corresponds to an RS-232C logical-1 (i.e., HIGH). When a TTL LOW is applied to V1, condition 3 is satisfied, so the output of the operational amplifier will be high positive (i.e., +8 to +10 VDC). This logicl level corresponds to the RS-232C LOW condition.

A TTL-to-RS-232C interface is shown in Figure 12-14A which does not depend upon an operational amplifier. When a HIGH is applied to the TTL input, the LED inside the optoisolator will be turned off. The switch Q1 is turned off, presenting a very high collector-emitter resistance. The voltage at the RS-232C output will be close to

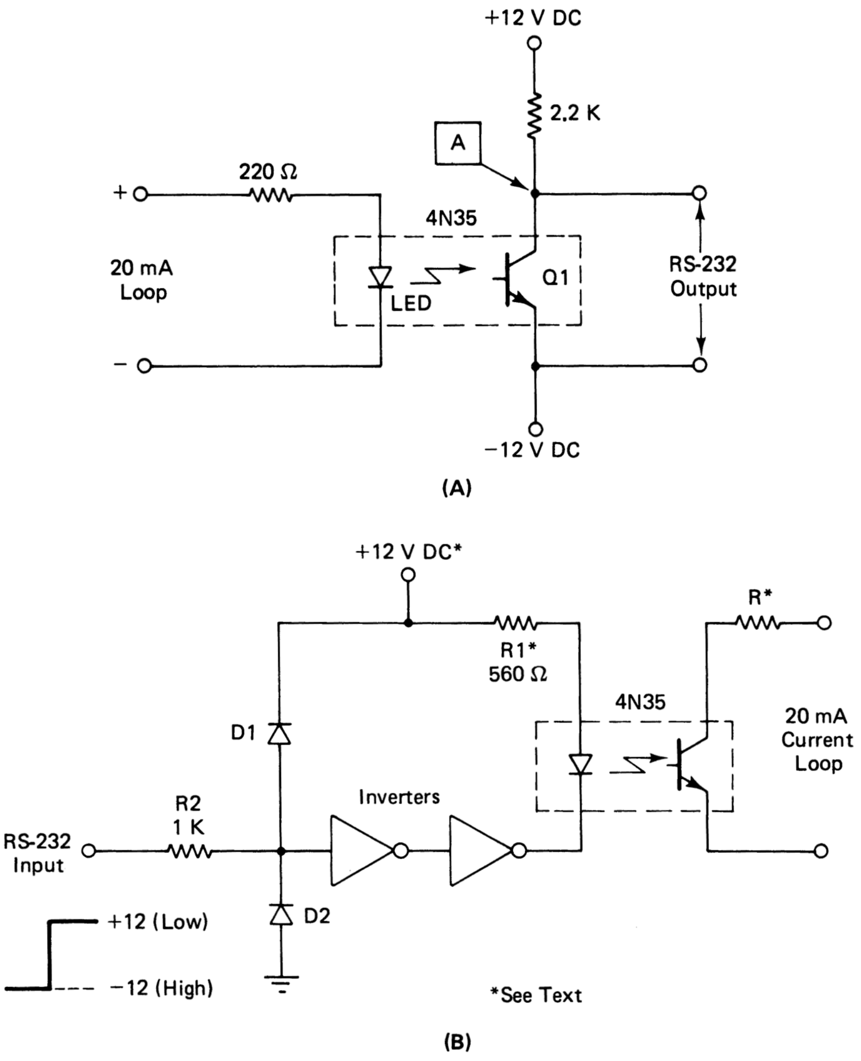


Figure 12-12. A) Isolated RS-232C output, B) isolated RS-232C input

−12 volts, which is the RS-232C HIGH condition. If, on the other hand, the TTL input is LOW, the LED is turned on, and the transistor is also on. The RS-232C output is now +12 VDC, which is the condition for LOW under the RS-232C convention.

A nonisolated TTL-to-RS-232C interface circuit is shown in Figure 12-14B. In this circuit, transistor Q1 is the switch that selects the +12 VDC or −12 VDC RS-232C logical levels, while Q2 controls Q1. Since Q1 is a PNP transistor, it will turn on when its base is more negative (or less positive) than its emitter. Therefore, when transistor Q2 is turned on, and its collector is at close to ground potential, then Q1 is also turned on. If Q2 is turned off, however, its collector potential rises to nearly +12 VDC, and Q1 is thereby turned off.

The key to switching between HIGH (−12 VDC) and LOW (+12 VDC) RS-232C levels, then, is to turn Q2 on and off. If Q2 is on, then the output is +12 VDC or LOW; if Q2 is off, then the output is −12 VDC or HIGH. If the TTL input is LOW, then the output of IC1 is HIGH, causing Q2 to be biased *on* through the 10 kohm resistor. Thus, a LOW on the TTL input turns on, producing a LOW on the RS-232C output. Similarly, a HIGH on the TTL input turns off Q2, thereby producing a HIGH on the RS-232C output.

Finally, we have the RS-232C-to-TTL interface circuit, shown in Figure 12-15. This circuit consists of a single transistor (Q1) and associated collector load (R2) and base bias (R1) resistor. When an RS-232C LOW (+12 VDC) is applied to the input, Q1 is turned on hard, so the TTL output is at ground potential. Thus, an RS-232C LOW at the input produces a TTL LOW on the output.

If an RS-232C HIGH (−12 VDC) is applied to the input, transistor Q1 is reverse biased, and is turned off. The collector potential rises to +5 VDC, which is the TTL HIGH. Diode D1 clamps the negative voltage to 0.7 VDC, which is safe for Q1.

When interfacing between TTL and RS-232C, do not overlook the possibility of using the Motorola MC1488 and MC1489 devices.

Controlling External Circuits

Control of external circuitry makes the microcomputer more useful. Certain calculation or signal processing chores can be performed in the machine, and then used to control external circuits. The simplest forms of external control are on-off switches that are controlled by a

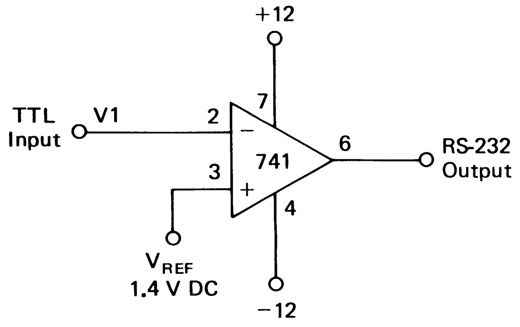
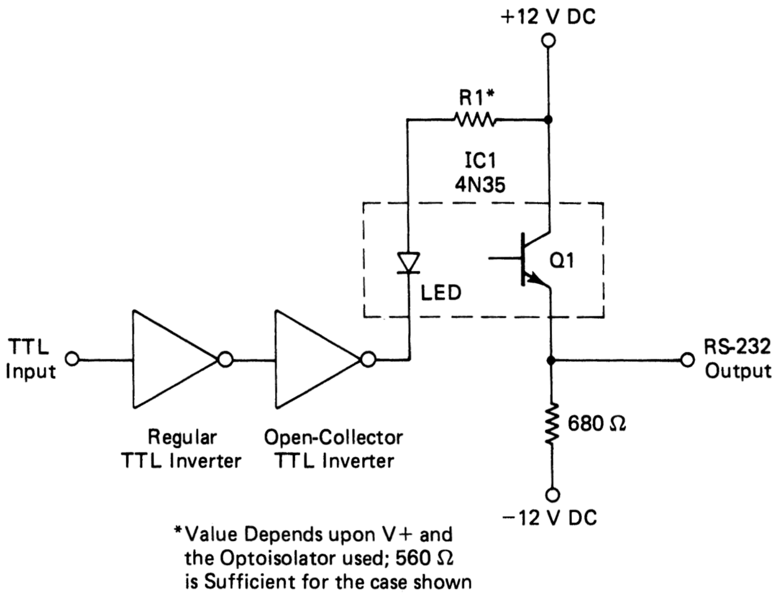
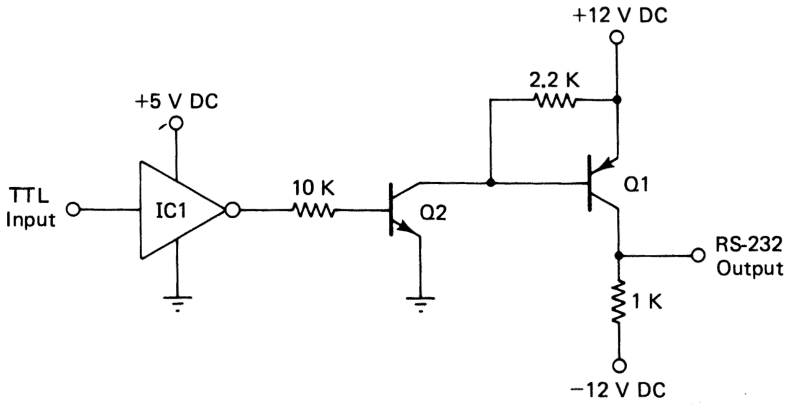


Figure 12-13. Op-amp forms TTL-to-RS-232 level translator


+12 V DC
R1*
IC1
4N35
Q1
LED
TTL Input
RS-232 Output
680 Ω
Regular TTL Inverter
Open-Collector TTL Inverter
−12 V DC
*Value Depends upon V+ and the Optoisolator used; 560 Ω is Sufficient for the case shown

(A)



(B)

Figure 12-14. TTL-to-RS-232 level translator A) isolated, B) nonisolated

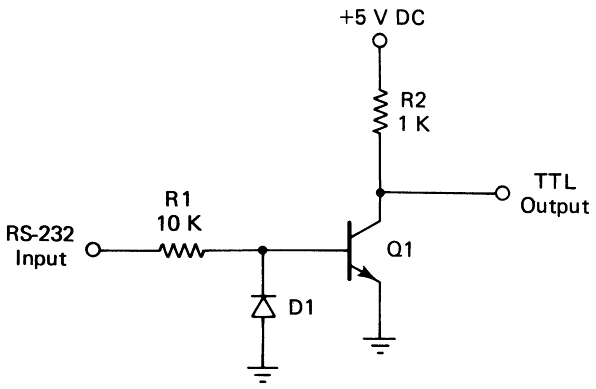


Figure 12-15. RS-232-to-TTL level translator

single bit of the computer's output port. More complex control applications will use devices such as amplifiers, digital-to-analog converts (DACs), etc. Extremely complex feedback control systems have been implemented using computers. The availability of microcomputers has only accelerated the process, and has, in an interesting way, made the design of computerized control circuits less a game for arcane areas of engineering and more a game for all.

Some external control circuits have already been discussed in Chapter 6, where we showed methods for connecting the computer to digital display devices such as the 7-segment LED decimal display. Some of the same methods are also used to interface other devices. For example, Figure 12-16 shows methods of interfacing electromechanical relays.

Why would we want to interface an electromechanical relay, which is a century-old device, to a modern space-age device like a microcomputer? The old relay may well be the best solution to many problems, especially where a certain degree of isolation is needed between the computer and the controlled circuit. An example might be 115-volt AC applications, especially those that may require heavy current loads. A typical "homeowner" application might be turning on and off 115-volt AC lamps around the house. The computer could be used as a timer and will turn on and off the lights according to some programmed schedule, for example, when you are away. Another application might be to use the computer to monitor burglar alarm sensors, and then turn on a lamp if one of them senses a break-in.

Figure 12-16 shows two basic methods for connecting the relay to the computer. Control over the relay is maintained by using 1 bit of the computer output port, in this case B0. Since only 1 bit is used, the other 7 bits are available for other applications, which may be displays, other relays, or certain other devices. Only 1 bit is used, so the others remain available and are not removed from use.

Most microcomputer outputs are not capable of driving heavy loads. Some devices will have a fan-out of 10 (i.e., will drive 18 milliamperes at +5 volts), while others have a low fan-out, typically 2 (3.6 mA). To increase the drive capacity and to provide a mechanism for control, we use an open-collector TTL inverter stage, U1. One end of relay coil K1 is connected to the inverter output, and the other end of the coil is connected to the V+ supply. Some TTL devices (7406, 7407, 7416, and 7417) will operate with potential greater than +5 volts DC on the output, so we can use 6-volt, 12-volt, or 28-volt relays. Of course, the package DC potential applied to the inverter is still the normal +5 volts required by all TTL devices. These inverters are actually hex inverters, so will contain 6 individual inverter circuits in each package. All 6 inverters can be operated independent of each other.

The operation of the circuit revolves around the fact that the relay (K1) coil is grounded when the inverter output is LOW, and ungrounded when the inverter output is HIGH. As a result, we can control the on-off states of the relay by applying a HIGH or LOW level to the input of the inverter. If the inverter input is LOW, for example, the output is HIGH so the relay coil is not grounded. In that case, the

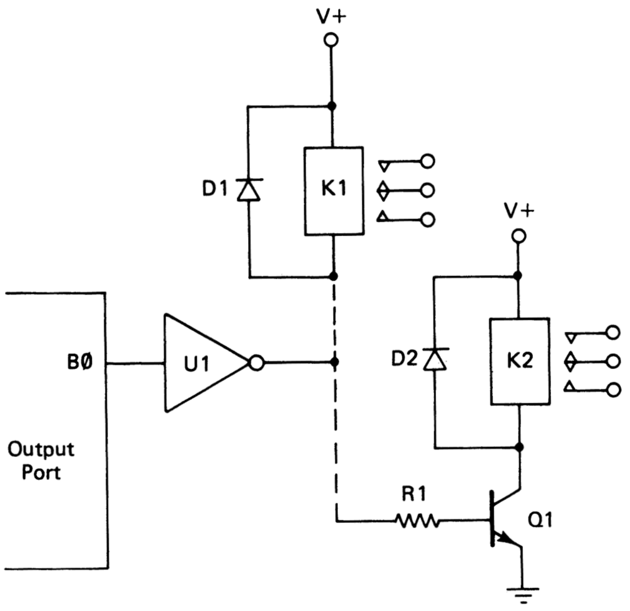


Figure 12-16. Computer interfacing with relays

relay coil is not energized because both ends are at the same electrical potential. When a HIGH is applied to the input of the inverter (i.e., when B0 of the output port is HIGH), then the inverter output is LOW and that makes the "cold" end of the relay coil grounded. The relay will be energized, closing the contacts. We may turn the relay on, then, by writing a HIGH (logical-1) to bit B0 of the output port, and turn it off by writing a LOW (logical-0) to the output port.

The inverter devices cited here have greater output current capability than some TTL devices, but are still low compared with the current requirements of some relays. High current relays, for example, may have coil current requirements of 1 to 5 amperes. If we want to increase the drive capability of the circuit, we may connect a transistor driver such as Q1 shown Figure 12-16.

In the case of relay K2, the cold end of the coil is grounded or kept high by the action of transistor Q1. This relay driver will ground the coil when the transistor is turned on (i.e., saturated), and will unground the coil when the transistor is turned off. As a result, we must design a method by which the transistor will be cut off when we want the relay off, and saturated when we want the relay on.

For circuits such as K2, the TTL interface with the computer output port (U1) may be an inverter or a noninverting TTL buffer. Of course, the on/off protocol will be different for the two. Also, we need not use an open-collector inverter for U1 as was the case previously. If we want to use an open-collector device, however, then we can supply 2.2 kohm pull-up resistor from the inverter output to the +5-volt DC power supply. The idea in this circuit is to use the inverter or buffer output to provide a bias current to transistor Q1. The value of the base resistor (R1) is a function of the Q1 collector current and the beta of Q1. This resistor should be selected to safely turn on the transistor, all the way to saturation, when the output of U1 is HIGH.

The relay will be energized when the output of Q1 is HIGH. Therefore, the B0 control signal should also be HIGH if U1 is a non-inverting buffer, and LOW if U1 is an inverter.

Both relays K1 and K2 in Figure 12-16 use a diode in parallel with the relay coil. This diode is used to suppress the so-called inductive kick spike created when the relay is de-energized. The magnetic field surrounding the coil contains energy. When the current flow is interrupted, the field collapses causing that energy to be dumped back into the circuit. The result is a high voltage counter-EMF spike that will possibly burn out the semiconductor devices or in the case of digital circuits, create "glitches"—pulses that shouldn't be! The diode should be a rectifier type with a peak inverse voltage rating of 1000 volts, and a current of 500 milliamperes or more. The 1N4007 diode has a

1000 PIV rating at 1 ampere. This diode will suffice for all but the heaviest relay currents.

Figure 12-17 shows a method for driving a relay from a low fan-out output port bit without the use of the inverter. The transistor driver is a pair of transistors connected in the Darlington Amplifier configuration. Such a circuit connects the two collectors together; the base of Q1 becomes the base for the pair; the emitter of Q2 becomes the emitter for the pair. The advantage of the Darlington Amplifier is that the current gain is greatly magnified. Current gain, *beta*, is defined as the ratio of the collector current to base current (I_c/I_b). For the Darlington Amplifier, the beta of the pair is the product of the individual beta ratings:

$$\beta_{1-2} = \beta Q1 \times \beta Q2$$

$$\beta_{1-2} = \beta^2$$

This equation is used when the two transistors are identical. Since the total *beta* is the product of the individual *beta* ratings, when two identical transistors are used, this figure is the *beta* squared.

You can either use a pair of discrete transistors to make the Darlington pair or use one of the newer Darlington devices that house both transistors inside one T0-5, T0-66, or T0-3 power transistor case.

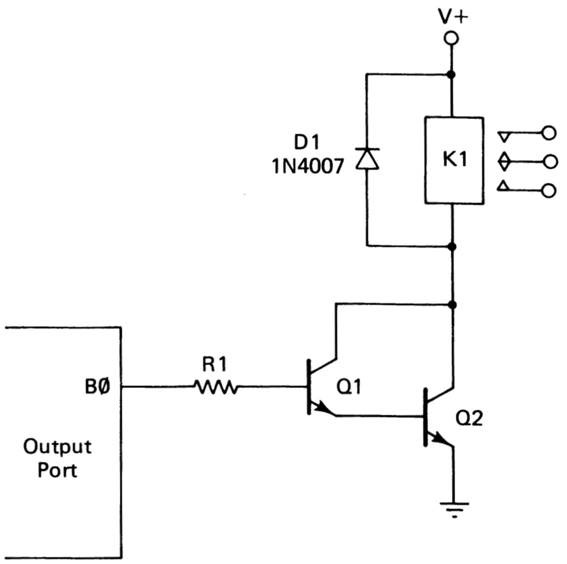


Figure 12-17. Interfacing with high current relays and solenoids

Another method for isolating dangerous or heavy-duty loads from the microcomputer output port was shown in Figure 12-8. In this case we use an optoisolator as the interface media. The optoisolator uses light flux between an LED and a phototransistor to couple the on-off signal from input to output. The LED produces light when a current is caused to flow in it, while the phototransistor is turned on (saturated) when light falls on the base and is off when the base is dark (LED is off). The transistor and LED are housed together, usually in a 6-pin DIP package.

The LED in the optoisolator is connected to the output of an open-collector TTL inverter. The cathode end of the LED is grounded, and the LED thereby is turned on whenever the output of the inverter is LOW. Thus, the LED is turned on whenever bit B0 of the output port is HIGH. At the instant the LED is turned on, transistor Q1 becomes saturated, so collector-emitter current flows in resistor R4, thereby causing a voltage drop that can be used for control purposes.

The voltage drop across resistor R4 can be used to drive another NPN transistor that actually controls the load. Or we can create an RC differentiator (R2/C1) and use the leading edge of the voltage across R4 (as it turns on) to trigger some other device. In Figure 12-18, for example, we are using a triac to control the AC load. A triac is basically a full-wave silicon controlled rectifier (SCR), and will gate-on when a pulse is received at the gate (G) terminal. Most triacs or SCRs will not turn off with gate signals, so some means must be provided to reduce the cathode-anode current to near zero when we want to turn off the device. That is the purpose of switch S1. When we want to turn the circuit off, switch S1 is opened long enough to allow the triac/SCR to revert to its off condition. Some devices allow turn-off as well as turn-on by external pulses.

A method for interfacing the microcomputer with display devices such as an oscilloscope or a strip-chart paper recorder is shown in Figure 12-19. In some instances, those devices are the most appropriate means of display, so we will want to provide some means to convert binary data to analog voltages for the 'scope or recorder. In Figure 12-19, the conversion is made by a digital-to-analog converter (DAC). The DAC produces an output potential V_o that is proportional to the binary output. Since various coding schemes are available, they will not be discussed here. We will assume for the purposes of our discussion that straight binary coding is used in which the zero-volts state is represented by a binary word of 00000000, and full-scale output is represented by the binary word 11111111. States in between zero and full-scale are represented by proportional binary words; half-scale, for example, is represented by 10000000.

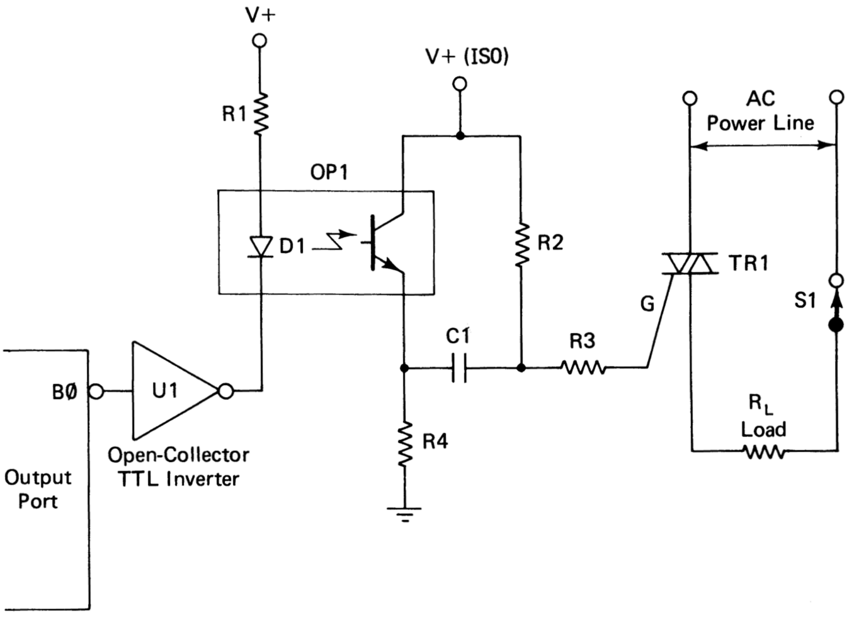


Figure 12-18. Isolated interface to control 120 VAC power lines

We will want to be able to scale the output potential V_{ot} to some value that is compatible with the display device. Not all oscilloscopes or paper recorders will accept any potential we apply, so some order must be introduced. Some oscilloscopes used in special medical, scientific, or industrial monitor applications, for example, come with fixed 1-volt inputs. Those instruments are often the most likely to be selected for applications involving a computer, yet lack the multi-voltage input selector of engineering models. For those we must select a DAC output voltage V_o that will match the 'scope input requirements. If the DAC output is somewhat higher (0 to 2.56 volts is common), then some form of output attenuation is needed. The operational amplifier used in Figure 12-19 provides that attentuation.

The voltage gain of an ordinary operational amplifier connected in the inverting follower configuration, as in the case of A1 in Figure 12-19, is set by the ratio of feedback to input resistances (i.e., R2 and R1). For this circuit, the gain is $(-R2/R1)$; the minus sign is an indication of polarity inversion. The inversion, incidentally, means that we must either design the DAC output to be negative or the oscilloscope/recorder input to be negative. We can re-invert the signal by following the amplifier with another circuit that is identical, except that R2 is a fixed resistor rather than a potentiometer. In that case, R1 = R2 = 10 kohms or any other value that is convenient. The product of two

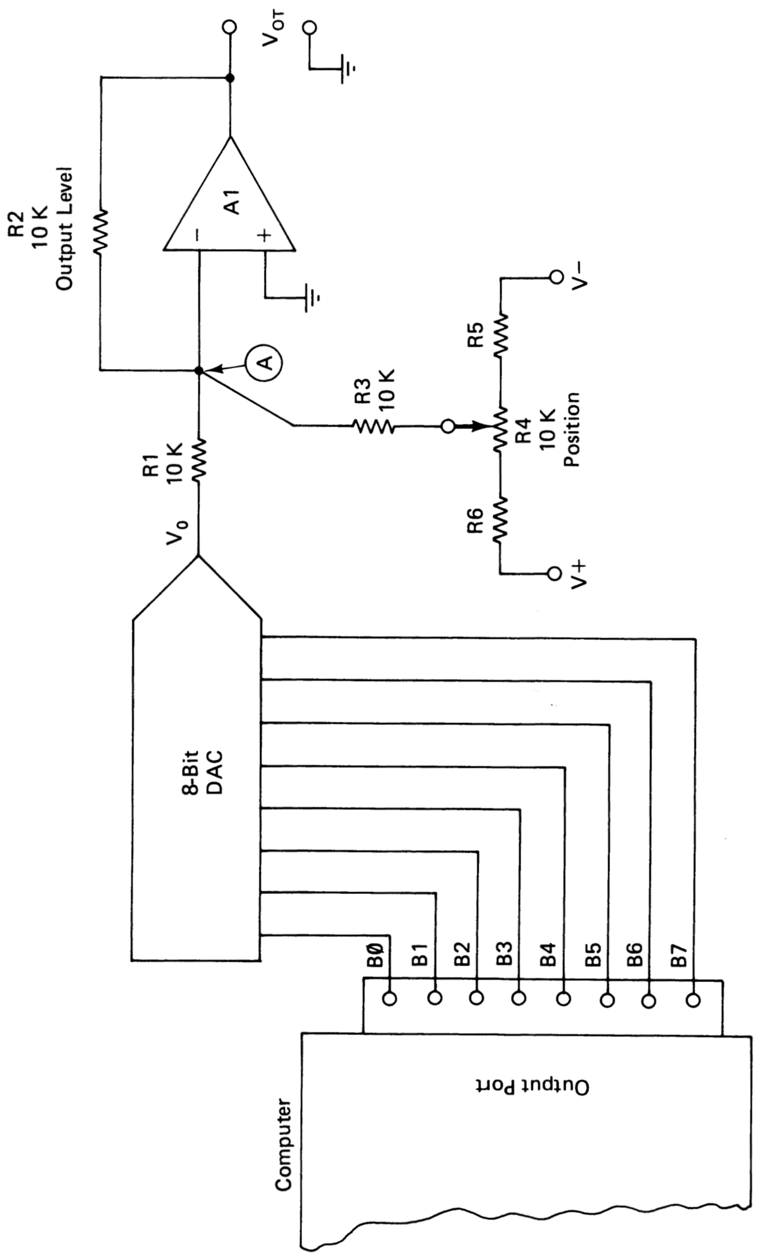


Figure 12-19. Digital-to-analog converter interface

inversions is the same as if none had taken place; V_{ot} will be in phase with V_o.

A position control is provided by potentiometer R4. In this circuit, we are producing an intentional output offset potential around which the waveform V_o will vary. The effect of this potential is to position the waveform on the oscilloscope screen or chart paper where we want it. Sometimes the baseline (i.e., zero-signal) position will be in the center of the display screen or paper, while in other cases it will be at one limit or the other.

An alternative system that would allow positioning of the baseline under program control is to connect a second DAC (with its own R1) to point A, which is the operational amplifier summing junction. The program can output a binary word other than DAC, which represents the desired position on the display. That position can be controlled automatically by the program or manually in response to some keyboard action by the operator. That approach requires the investment of one additional DAC, but that cost is now no longer so horrendous as it once was—IC DACs are almost dirt cheap these days.

If the DC load driven by the DAC/computer combination is somewhat more significant than an oscilloscope input, then the simple op-amp method shown in Figure 12-19 may not suffice. For those applications we may need a power amplifier to drive the load.

A power amplifier is shown in Figure 12-20. Here we have a complementary symmetry class-B power amplifier. A so-called "complementary pair" of power transistors is a pair, one NPN and the other PNP, that are electrically identical except for polarity. When these transistors are connected with their respective bases in parallel, and their collector-emitter paths in series, the result is a simple push-pull class-B amplifier. When the DAC output voltage V_o goes positive, then transistor Q1 will tend to turn on, and current flowing through Q1 under the influence of V+ will drive the load also positive. If, on the other hand, the output voltage of the DAC is negative, then PNP transistor Q2 will turn on and the load will be driven by current from the V− power supply. Since each transistor turns on only on one-half of the input signal, the result is fullwave power amplification when the two signals are combined in the load.

The "load" in Figure 12-20 can be any of several different devices. If it is an electrical motor, for example, the DAC output voltage will vary the speed of the DC motor, hence the computer will control the speed because it controls V_o. If we provide some means for measuring the speed of the motor, then the computer can be used in a negative loop to keep the speed constant, or change it to some specific value at will.

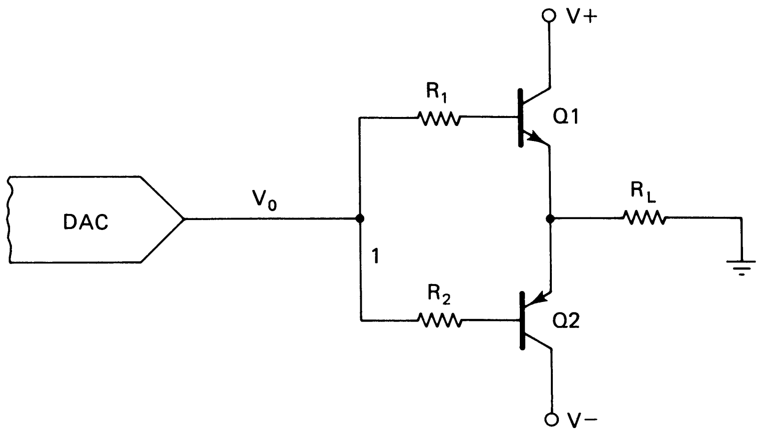


Figure 12-20. High current DAC output

A method exists by which the motor can be controlled without the DAC. If we use a transistor driver to turn the motor on and off, we can effectively control its speed by controlling the relative duty cycle of the motor current. By using a form of pulse width modulation, we can set the motor speed as desired.

Pulse width modulation of the motor current works by setting the total percentage of unit time that the motor is energized. The current will always be either all on or all off, never at some intermediate value. If we vary the length of time during each second that current is applied, therefore, we control the total energy applied to the motor, hence its speed. If we want the motor to turn very slowly, then we arrange to output very narrow pulses through the output port to U1 to the motor control transistor. If, however, we want the motor speed to be very fast, then long-duration pulses, or a constant level, are applied to the output port.

Can you spot the most common programming error that will be made when you actually try to implement this circuit? It occurs at turn-on. The DC motor has a certain amount of inertia that keeps it from wanting to start moving when it is off. As a result, if we want to start the motor at a slow speed, then the pulse width may not be great enough to overcome inertia, and the motor will just sit dormant. The solution is to apply a quick, one-time, long-duration pulse to get the motor in motion any time we ask it to turn on from a dead stop. After the initial "kick in the pants," the normal pulse coding will apply.

If we want to actively control the speed of the motor, then we will need some sort of sensor that converts angular rotation into some kind of pulse train. On some motors this problem is less of a nuisance because the motor is mechanically linked with an AC alternator housed in the same case. There will be a pair of output terminals that exhibit an AC sinewave whenever the motor shaft is rotating. If we apply this AC signal to a voltage comparator (such as the LM-311 device), then we will produce a TTL-compatible output signal from the comparator that has the same frequency as the AC from the motor. A typical case uses the inverting input of the comparator to look at the AC signal, and the noninverting input of the comparator is at ground potential. Under this condition, an output pulse will be generated every time the AC signal crosses the zero-volt baseline. Such a circuit is called a zero-crossing detector, appropriately enough.

If there is no alternator, then some other means of providing the signal must be designed. One popular system is shown in Figure 12-

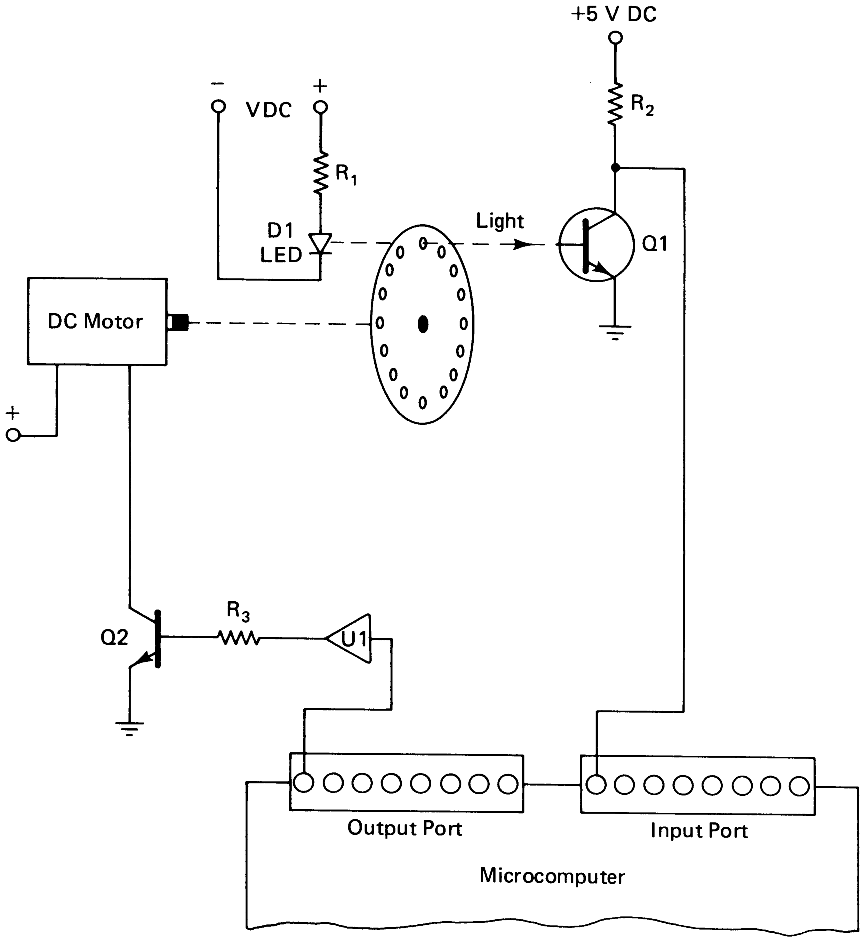


Figure 12-21. Motor control example

21, in which a wheel with holes in the outer rim is connected to the motor output shaft, a light emitting diode (LED), and phototransistor whenever a hole in the wheel is in the path. Otherwise the light path is interrupted. Flashes of light produced when the wheel rotates trigger the transistor to produce a signal that is, in turn, applied to the input port bit as shown. A program can then be written to sample this input port bit, and then determine the motor speed from the frequency of the pulses, or, as is more likely with some microprocessors, the time between successive pulses.

The sensor shown in Figure 12-21 may be constructed from discrete components, if desired, but be aware that several companies make such sensors already built into a plastic housing. A slot is provided to admit the rim of the wheel to interfere with the light path.

The methods shown in this chapter are intended to be used as guides only, and you may well come up with others that are a lot more clever. The computer doesn't need much in the way of sophisticated interfacing in most cases, as can be seen from some of these examples.

13 Interrupts

The *interrupt* function on any computer allows external devices to gain control of the computer. We might want to have such a capability for several reasons. First, we might want to permit the computer to do some other job while awaiting some alarm condition or another. For example, the computer can be used as an environmental controller in a home or business. Under most circumstances, the computer would monitor temperature and humidity levels, and control heaters, air conditioners, and humidifiers. But, if a fire or intruder sensor becomes active, forcing an interrupt, then the computer will cease executing the normal program and switch to the subroutine that serves that type of alarm.

Another case might be to interface with peripherals that are either too slow for the computer or only operate occasionally. Printers, especially older mechanical teletypewriters, are particularly slow. We find these devices are so slow that a 1-mHz 6502 can execute thousands of instructions during the time required for sending one character to a printer, i.e., about 100 milliseconds. We can, however, write a program that will output a character and then go do something else until the printer sends back a ready signal that interrupts the 6502.

Still another case involves devices such as A/D converter interfaces. Such a device will produce an n-bit binary word that is proportional to some analog input voltage. Some A/D converters require tremendous chunks of time to make the conversion. Some dual-slope integration types, for example, require 50 milliseconds. We can, however, use the end of conversion (EOC)—also called status or data ready—to interrupt the computer. That arrangement permits the computer to perform other chores, for example, process the A/D converter data, while the A/D is "doing its thing."

There are two interrupt lines on the 6502, $\overline{\text{IRQ}}$ and $\overline{\text{NMI}}$. Both of these pins are active-LOW TTL-compatible lines. This means that they are LOW when the applied voltage is 0 to 0.8 volt, and HIGH when the applied voltage is +2.4 volts or more.

There is a major difference between the two forms of interrupt. The $\overline{\text{NMI}}$ is a nonmaskable interrupt. When NMI goes LOW, the computer must go to the interrupt service routine. The $\overline{\text{IRQ}}$, or interrupt request line, is maskable. This interrupt request will be honored only if the IRQ Disable (I-flag) bit in the 6502 processor status register is reset (LOW). If the I-flag is HIGH, then the 6502 will not honor an interrupt request on the $\overline{\text{IRQ}}$ line.

There are two ways to set the I-flag. First, we can execute a software SEI (set interrupt disable status) instruction. The result of SEI is to set I = 1. The other way to set the I-flag is to reset the computer. When the $\overline{\text{RST}}$ line on the 6502 is brought LOW, the processor jumps to a location set by a vector stored in page-FF of memory. During the execution of this operation, the I-flag is set HIGH.

The only way to reset the I-flag, thereby permitting interrupts on $\overline{\text{IRQ}}$, is to execute a CLI (clear interrupt disable status) instruction. When the CLI instruction is completed, the I-flag will be LOW. This condition permits the 6502 to respond to interrupt requests.

One implication of the above discussion is that the programmer must permit the $\overline{\text{IRQ}}$ line to be active. Almost all computers generate a power-on reset pulse that momentarily brings $\overline{\text{RST}}$ LOW immediately after power is applied to the system. Thus, the I-flag is set HIGH, disabling $\overline{\text{IRQ}}$, every time (1) power is applied, or (2) the operator presses a reset button. If the program is to respond to interrupt requests on $\overline{\text{IRQ}}$, then the programmer must initialize the system by executing $\overline{\text{CLI}}$ sometime prior to the time when interrupts are being sought. In many cases, this chore is done when the program first begins execution. There may also be times we want the program to turn the I-flag on and off in response to different conditions.

INTERRUPT VECTORS

A vector is an operand stored at a specific location that is used to alter the contents of the program counter. In 6502-based systems, the vectors are stored in page-FF of memory, as follows:

FFFAH	$\overline{\text{NMI}}$ low address byte
FFFBH	$\overline{\text{NMI}}$ high address byte
FFCH	reset low address byte

FFFDH	reset high address byte
FFFEH	$\overline{\text{IRQ}}$ low address byte
FFFFH	$\overline{\text{IRQ}}$ high address byte

If a nonmaskable interrupt request occurs, then the 6502 goes to location FFFAH and fetches the low address byte and places it in the low-order half of the program counter (PCL). It then goes to location FFFBH and fetches the high address byte and stuffs it into the high-order half of the program counter (PCH). The address of the next instruction to be executed will be (PCH + PCL). This address is the beginning instruction of the interrupt service subroutine. The last instruction in the subroutine must be RTI (return from interrupt). When this instruction is encountered, the program counter will be loaded with the address of the next instruction in the main program that would have been executed if no interrupt had occurred.

NONMASKABLE INTERRUPTS

The nonmaskable interrupt does not depend upon the condition of the I-flag in the processor status register. When the $\overline{\text{NMI}}$ line goes LOW, the 6502 will jump to the nonmaskable interrupt subroutine, as directed by the vector addresses stored at FFFAH and FFFBH. The jump occurs when the instruction being executed (when $\overline{\text{NMI}}$ goes LOW) is completed. Following execution of the interrupt subroutine, as the RTS instruction is executed, the program jumps back to the next sequential instruction of the main program.

Figure 13-1 diagrams the operation of the 6502 during a nonmaskable interrupt. In this hypothetical example, the computer is executing a program in page-03 when, at location 0353H, $\overline{\text{NMI}}$ signal is received (i.e., $\overline{\text{NMI}}$ goes LOW). The operation is as follows:

1. An active (LOW) $\overline{\text{NMI}}$ is received while the 6502 is executing an instruction at 0353H.
2. At the finish of executing the instruction at 0353H, the 6502 goes to FFFAH and fetches the low-order address byte (00H) and places it in the PCL half of the program counter. It then goes to FFFBH and fetches the high-order address byte (C0H) and places it in the PCH half of the program counter.
3. The contents of the program counter are not C000H, so the 6502 goes to that location to pick up the first instruction of the NMI service subroutine.
4. At the end of the service subroutine, an RTI instruction is encountered. The address of the next instruction of the main

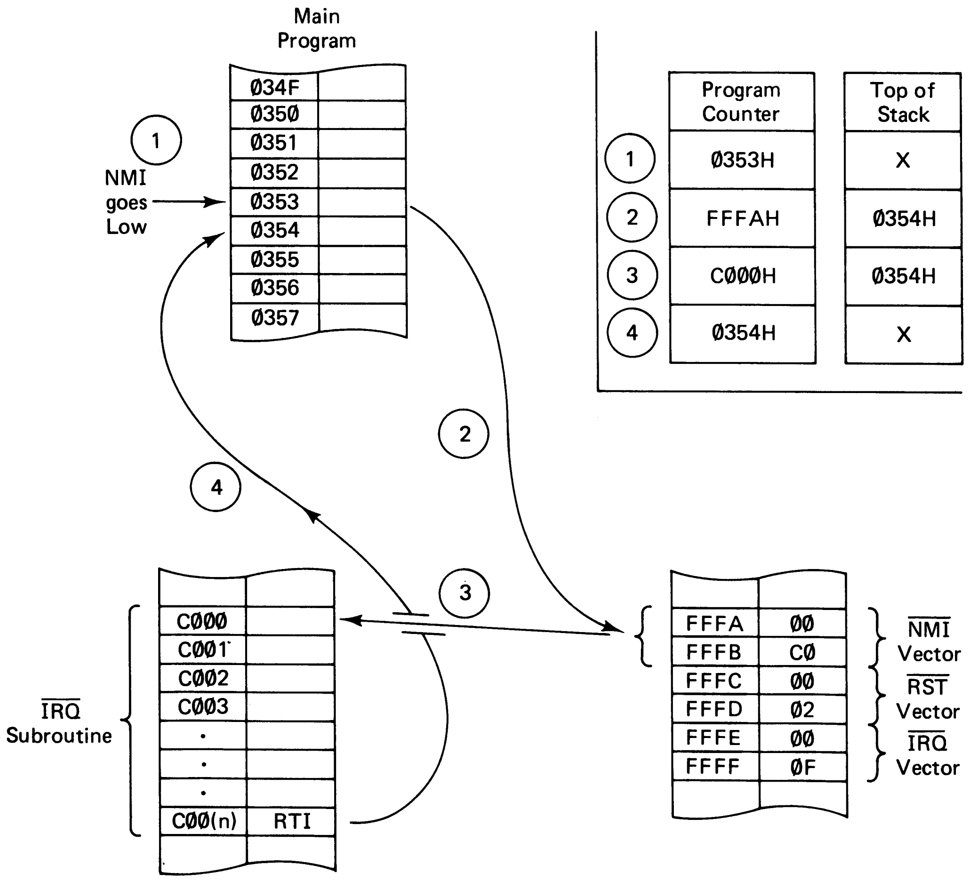


Figure 13-1. Nonmaskable interrupt sequence example

program (0354H) is recovered from the stack in page 0. Control is returned to the main program at location 0354H.

The nonmaskable interrupt is used where the system cannot tolerate ignoring an interrupt, for example, a critical alarm in a life-threatening situation. The programmer cannot disable the $\overline{\text{NMI}}$ line. The I-flag is disabled during the execution of the service subroutine, and it will be re-enabled during RTI execution (provided that the I-flag was enabled at the beginning of the subroutine).

MASKABLE INTERRUPT REQUESTS

The interrupt request ($\overline{\text{IRQ}}$) is an active-LOW 6502 input which operates in a manner similar to the nonmaskable interrupt discussed previously. The differences between $\overline{\text{IRQ}}$ and $\overline{\text{NMI}}$ are in the use of the flag. Also, the $\overline{\text{NMI}}$ will be recognized if it is LOW for at least two clock cycles, while $\overline{\text{IRQ}}$ must be held LOW until it is recognized. Most devices connected to the interrupt request line will have a flip-flop output which can be reset under program control. It is common practice to clear the interrupt request (cancelling the request) under program control as part of the service subroutine. The 6502 sets the I-flag HIGH when it responds to an $\overline{\text{IRQ}}$ so that the machine won't respond to the same interrupt more than once. The program must also reset the I-flag by executing a CLI command if the intent is to respond to eventual interrupt requests. Thus, the interrupt service subroutine must (1) set any external interrupt flip-flops HIGH, and (2) execute a CLI instruction to clear the I-flag. Most progammers prefer to perform these chores immediately before the RTI (return from interrupt).

Figure 13-2 shows the operation of the interrupt request ($\overline{\text{IRQ}}$) line when the I-flag in the PSR is set (HIGH). The 6502 is executing an instruction in the main program at location 0353H when $\overline{\text{IRQ}}$ goes LOW. When the 6502 has finished executing the instruction at 0353H, it tests the I-flag in the PSR (see step 2 in Figure 13-2). Since the I-flag is HIGH, the 6502 sees that $\overline{\text{IRQ}}$ is to be ignored. Thus the program counter is updated to the next step in the main program (0354H) rather than the $\overline{\text{IRQ}}$ vector. The program will continue executing as if no interrupt request had occurred.

Operation of $\overline{\text{IRQ}}$ when the I-flag is LOW is shown in Figure 13-3. This condition indicates that the interrupt line is not disabled. Again, the scenario is the same; the 6502 is executing an instruction at 0353H in the main program when $\overline{\text{IRQ}}$ goes LOW. The following sequence ensues:

1. An active (LOW) $\overline{\text{IRQ}}$ is sensed while the 6502 is executing an instruction at 0353H.
2. When the 6502 is finished executing the instruction at 0353H, it tests the I-flag in the processor status register (PSR) for HIGH or LOW.
3. Finding the I-flag LOW, the 6502 goes to location FFFEH to fetch the low address byte, and load it into the low-order half (PCL) of the program counter. It then goes to FFFFH to fetch the high address byte and store it in the high-order half of the program counter (PCH).
4. The address in the program counter is (PCH + PCL), so the 6502 jumps to address 0F00H, which is the starting address of the interrupt service program. During this period, the 6502

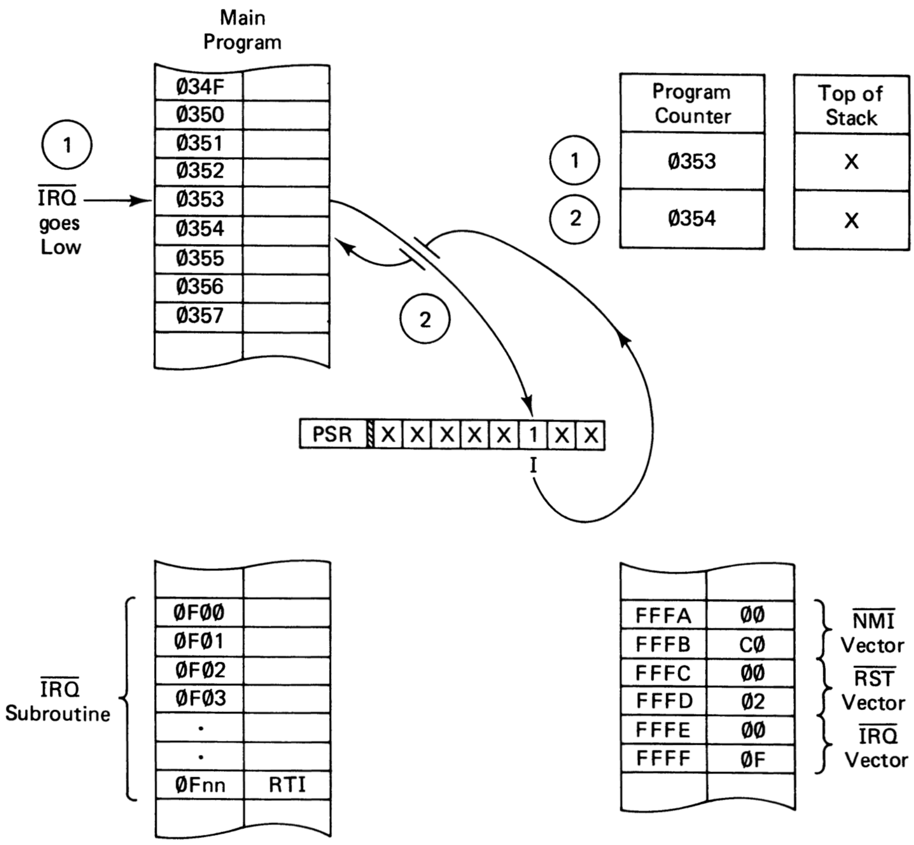


Figure 13-2. Maskable interrupt sequence example (masked)

has stored on the external stack the address of the location where the main program will resume (in this case, 0354H).

5. At the end of the interrupt program, the 6502 encounters an RTI (return from interrupt) instruction. This causes it to pop the address of the next instruction (0354H) off the stack and load it into the program counter. The main program resumes at 0354H.

If the programmer makes no provision for clearing the I-flag some time during the subroutine or subsequently on the main program, then the 6502 will no longer respond to interrupt requests.

RESET LINE AS INTERRUPT

The reset input ($\overline{\text{RSI}}$) of the 6502 is essentially a special limited form of nonmaskable interrupt. When $\overline{\text{RST}}$ drops LOW, the 6502 goes to the location indicated by the contents of FFFCH and FFFDH; the low-order byte of the 2-byte address is stored at FFFCH, and the high-order byte is at FFFDH. The main purpose of RST is to initialize the computer. A reset pulse is generated when power is first applied to the computer, and this action forces the computer to begin executing the program at the address stored in the reset vector, FFFCH/FFFDH. Normally, this vector address is the initial address of the program. If that is where you want to transfer program control for some particular class of interrupt, then you can "bootleg" an interrupt using the $\overline{\text{RST}}$ input.

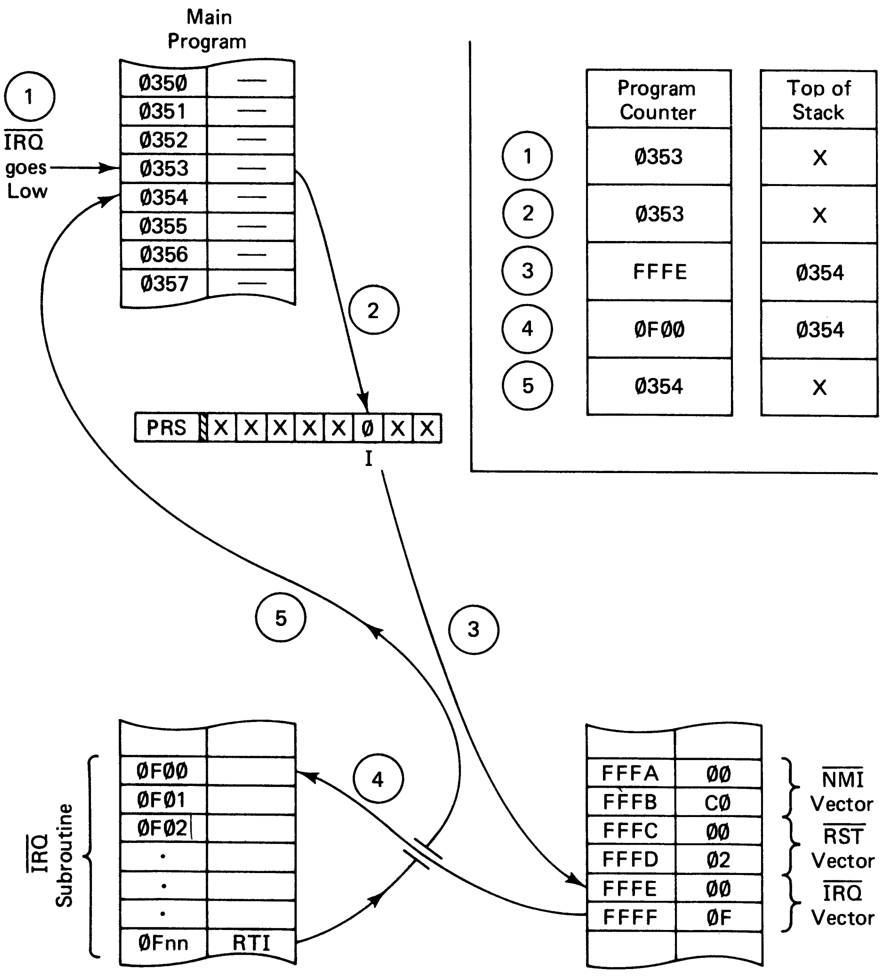


Figure 13-3. Maskable interrupt sequence example (unmasked)

MULTIPLE INTERRUPTS ON 6502

Unlike other microprocessors (e.g., Z-80), the 6502 has only one maskable and one nonmaskable interrupt line and/or mode. Without external circuitry, therefore, we can service only one device on each interrupt input. There are special chips available which permit us to add and/or prioritize multiple interrupts, but in this section we will examine two simple schemes to accomplish this same job with discrete logic elements.

The method shown in Figure 13-4 permits us to have up to 8 interrupts, and also permits us to software prioritize according to importance. Obviously, if we receive two interrupts, one indicating a fire and the other indicating that a new hot water temperature has been commanded, we want the computer to look at the fire alarm first.

Figure 13-4 connects to the computer via an I/O port, here designated as port-1 for the sake of convenience. There are 8 interrupt lines, designated as $\overline{\text{INT0}}$ through $\overline{\text{INT7}}$, all of which are active-LOW. Since the interrupt may be transient in nature, and the 6502 needs to see a LOW level rather than a negative-going edge, we provide flip-flops FF1–FF8 to "remember" the interrupt request until the 6502 recognizes it. Only two interrupt flip-flops are shown, again for the sake of simplicity.

When any interrupt request line goes LOW momentarily, the Q-output for its flip-flop will go HIGH (Note: The interrupt lines $\overline{\text{INT0}}$–$\overline{\text{INT7}}$ are connected to the active-LOW *set* inputs on the flip-flops). Each Q-output is connected to a single bit of the input port, and also to 1 input of an 8-input NOR gate. When any input of NOR gate G1 is HIGH, the output of G1 will be LOW. We can, therefore, use the output of G1 to signal either $\overline{\text{INT}}$ or $\overline{\text{NMI}}$ on the 6502, as appropriate.

But how does the 6502 know which interrupt service? Except for the case where any and all interrupts are served by the same program, we must have some means for distinguishing among the 8 different interrupt lines. That is the function of the input port.

When $\overline{\text{INT}}$ goes LOW, the 6502 will branch to the program whose initial address is stored at FFFEH and FFFFH. The first instruction in this subroutine will be to read input port-1 to determine which bits are HIGH (active) and which are LOW (inactive). The program will then clear the active flip-flops and then go to the portion of the program segment that serves the active interrupt request lines.

The programmer can prioritize the interrupt requests by causing the computer to respond to certain requests first, or by polling the bits in order of highest-to-lowest priority.

Resetting the interrupt flip-flops occurs when the clear line is brought LOW. Since each FF clear line is connected to an output port

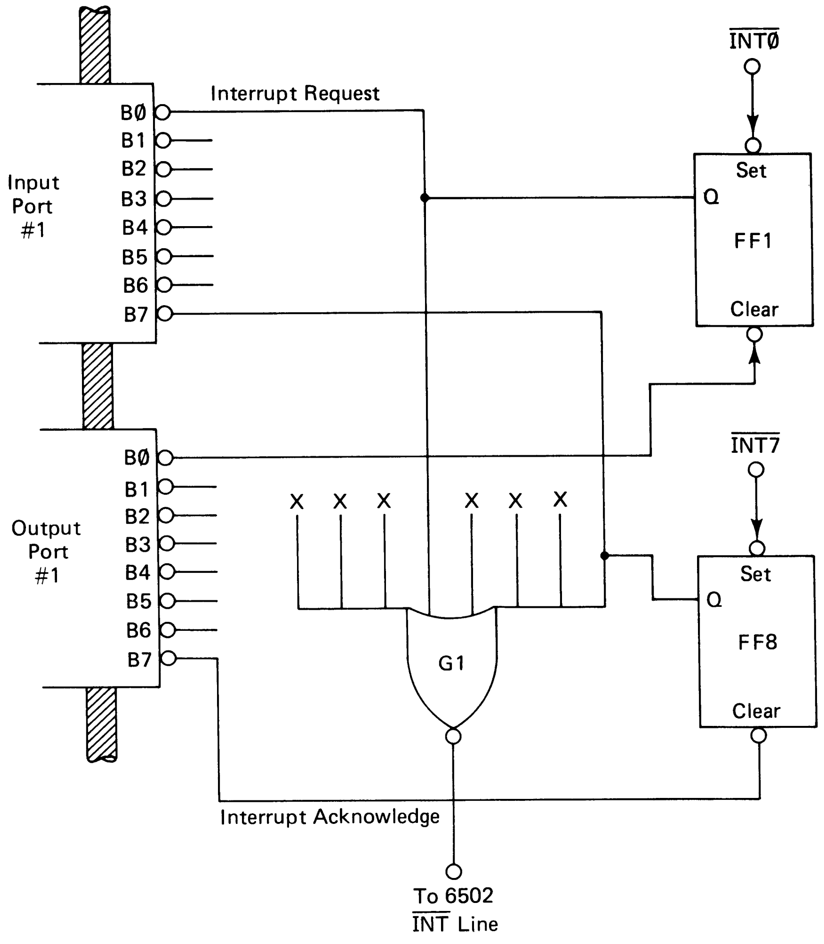


Figure 13-4. Multiple interrupt hardware

bit, we can reset, or "clear," the interrupt by writing a LOW to the correct output port bit. During the initialization sequence, right after the computer is powered-up (or a reset button is pressed), we may want to ensure that all interrupt lines are cleared by writing 00H to port-1 for a few milliseconds, and then following the FFH. This action will force all flip-flop clear lines LOW, forcing each Q-output LOW, and then setting all clear lines HIGH (the inactive state).

In cases where the computer and the peripherals do not share a common printed wiring board (or even the same cabinet), the interrupt flip-flops are usually located in the peripheral's circuitry, while gate G1 and the I/O ports are with the computer. The input port lines become interrupt request signals, while the output port lines are interrupt acknowledge signals.

14 Interfacing with the Apple II Bus

The Apple II microcomputer is probably one of the most popular complete microcomputers on the personal computer market. It is used by hobbyists, businesses, computer education instructors, and scientists. It is a powerful little machine, and is available in several versions, such as the Apple® II Plus and the enhanced Apple® IIe, which uses LSI microcircuits to replace many of the microcircuits which were found in the II and II Plus versions. The Apple II is found almost everywhere, and it seems that there are more Apple retailers than for any other computer except, perhaps, the TRS-80® by Radio Shack.

The Apple II is a self-contained microcomputer that is based on the 6502 microprocessor chip. This computer comes complete with a keyboard and up to 48K of internal random access read/write (RAM) memory chips. The 6502 will support 64K, but the Apple II uses the upper 16K for its own reserved purposes. Nevertheless, there are ways around this limitation (which is often more imagined than real), and some manufacturers offer Apple II 16K cards which bring the memory size up to the full 64K. It is necessary to use either programming or hardware tricks to let the computer use either the built-in read only memory (ROM) or the add-on RAM when addressing the upper 16K of memory.

The Apple II has been around a long time, and is now well-entrenched as one of the basic microcomputers. One advantage of this type of computer is that large amounts of software and hardware accessories are on the market for it. There are many imitators of the Apple II, most of which are software compatible with the Apple II for obvious reasons! But the Apple II is bedeviled with not only imitators, some of which use seemingly exact copies of the Apple II printed wiring board layout, but also counterfeits. Some unscrupulous manufacturers in Southeast Asia have offered for sale exact duplicates of the Apple II without first bothering with the legal nicety of a license from the U.S. manufacturer!

The Apple II is a single-board computer housed in a small case about the size of a cheap typewriter. There are eight slots on the motherboard that will accommodate accessories and interface devices. The basic computer comes with 16K of memory, but we can configure it with up to 48K of 8-bit memory by replacing the 4K memory chips with 16K memory chips.

The Apple II uses software to replace hardware complexity. The memory allocations above the 48K boundary are used for the monitor program and for housekeeping functions, such as driving the disk system.

The connectors for each of the plug-in cards have 50 pins, with pins 1 through 25 on the component side of the inserted printed wiring boards, and 26 through 50 on the "foil" side of the card. Several companies offer either plug-in accessory cards (I/O cards or A/D converter cards), or blank interfacing cards on which you may build your own circuitry. The Apple II plug-in card pinouts are described here:

Pin	*Designation*	*Function*
1	$\overline{\text{I/O SELECT}}$	This active-LOW signal is LOW if and only if one of the 16 addresses assigned to that particular connector is called for in the program. The 6502 used in the Apple II uses memory-mapped I/O, so each I/O port number is represented by a memory location in the range C800H and C8FFH. Reference the Apple II memory-map in the manual for specific locations.
2	A0	Address Bus bit 0
3	A1	Address Bus bit 1
4	A2	Address Bus bit 2
5	A3	Address Bus bit 3
6	A4	Address Bus bit 4
7	A5	Address Bus bit 5
8	A6	Address Bus bit 6
9	A7	Address Bus bit 7
10	A8	Address Bus bit 8
11	A9	Address Bus bit 9
12	A10	Address Bus bit 10
13	A11	Address Bus bit 11

Pin	*Designation*	*Function*
14	A12	Address Bus bit 12
15	A13	Address Bus bit 13
16	A14	Address Bus bit 14
17	A15	Address Bus bit 15
18	$R/\overline{W}$	Control signal from 6502 microprocessor is HIGH during read operations, and LOW during write operations.
19	(NC)	No connection
20	$\overline{I/O\ STR}$	Active-LOW signal that lets the world know that an input or output operation is taking place, this line will go LOW whenever an address in the range C800H to C8FFH is on the address bus.
21	$\overline{RDY}$	Active-LOW input, if this line is LOW during the phase-1 clock period, then the CPU will halt (i.e., enter a wait state) during the following phase-1 clock period. If $\overline{RDY}$ remains HIGH, then normal instruction execution will occur on the following phase-2 clock signal.
22	$\overline{DMA}$	Active-LOW Direct Memory Access line allows external devices to gain access to the data bus and apply an 8-bit data word to the address it places on the address bus.
23	INTOUT	Interrupt output signal allows prioritizing of interrupts from one plug-in card to another. The INTOUT line of each lower order card runs to the INTIN pin of the next card in sequence (see pin 28).
24	DMAOUT	Direct Memory Access version of INTOUT
25	+5	+5-volt DC power supply available from main board to plug-in card
26	GND	Ground
27	DMAIN	Direct Memory Input signal allows prioritizing DMA functions.
28	INTIN	Interrupt Input (see DMAOUT, pin 24)
29	$\overline{NMI}$	Active-LOW nonmaskable interrupt line. When brought LOW, this line will cause the CPU to be interrupted at the completion of the present instruction cycle. This interrupt is not dependent upon the state of the CPU's interrupt flip-flop flag.

Pin	*Designation*	*Function*
30	$\overline{\text{IRQ}}$	Interrupt Request. This active-LOW input will cause the CPU to interrupt at the end of the present instruction cycle, provided that interrupt flip-flop is reset.
31	$\overline{\text{RES}}$	Reset line. This active-LOW input will cause the program to return to the Apple II monitor program.
32	$\overline{\text{INH}}$	Active-LOW input that disconnects the ROMs of the monitor to permit custom software stored in ROMs on the plug-in board to be executed.
33	−12	−12-volt DC power from main board to plug-in board
34	−5V	−5-volt DC power from main board to plug-in board
35	(NC)	No connection
36	7M	7 mHz clock signal
37	Q3	2 mHz clock signal
38	01	Phase-1 clock signal
39	USER1	Similar to $\overline{\text{INH}}$ except that it disables all ROMs including C800H to C8FFH used for I/O functions
40	02	Phase-2 clock signal
41	$\overline{\text{DEVICESEL}}$	Active-LOW signal indicates one of the 16 addresses assigned to that connector is being selected.
42	D7	Data Bus bit 7
43	D6	Data Bus bit 6
44	D5	Data Bus bit 5
45	D4	Data Bus bit 4
46	D3	Data Bus bit 3
47	D2	Data Bus bit 2
48	D1	Data Bus bit 1
49	D0	Data Bus bit 0
50	+12	+12-volt power from main board to plug-in boards

15 Interfacing with the KIM-1, AIM-65, and SYM-1

The KIM-1 microcomputer was a single-board trainer that was introduced by MOS Technology, Inc. of Norristown, PA, the originator of the 6502 microprocessor chip. It was apparently intended to introduce the world of microprocessing to engineers who would incorporate the 6502 into their instrument and computer designs. The KIM-1 computer, however, blossomed into a popular starter computer as well as a trainer. Many current computer experts began their careers with a KIM-1 device.

The KIM-1 was a single-board computer that contained 1K of 8-bit memory, a 6522 Versatile Interface Adapter (VIA), a 20 mA TTY current loop for making hard copies, and a cassette (audio) interface to allow storage of programs on ordinary audio tape. One feature of the KIM-1 tape interface not found on others of the era is the ability to search for programs on the tape by a designator applied to the beginning of the program on the cassette.

The SYM-1 is a more recent single-board trainer computer that uses the KIM-1 bus. The SYM-1, however, is still easily obtained and contains more features than the original KIM-1. For those members of the KIM-cult, the SYM-1 is a good substitute.

The AIM-65 is a more advanced microcomputer based on the KIM bus, and is made by Rockwell Microelectronics, Inc. The AIM-65 computer uses a standard ASCII typewriter keyboard instead of the hexadecimal pad of the KIM-1. It also has a 20-character 5 × 7 dot matrix LED display and a 20-column 5 × 7 dot matrix thermal printer instead of the standard 7-segment LED readouts of the KIM-1, which require some imagination to read hexadecimal digits above 9. The printer uses standard calculator printer paper available at stationery stores.

The AIM-65 also has a sophisticated monitor program stored in ROM, and has the ability to incorporate BASIC and a 6502 assembler into other on-board ROMs. In contrast, the KIM-1 originally used a relatively simple monitor. To write and input programs one had to "fingerbone" instructions into the computer on a step-by-step basis. The AIM-65 comes with a text editor. Also, the AIM-65 can be configured with either 1K or 4K of memory, and external memory to 48K can be added.

The two interfacing connectors etched onto the boards of the KIM-1, SYM-1, and AIM-65 computers are the applications connector, basically an I/O connector, and the expansion connector, which is more similar to a genuine bus connector. Both are of primary interest to microprocessor users who must interface the computer with some external device.

KIM-1/SYM-1/AIM-65 Applications Connector

Note: Numbered connector pins are on the top—component—side of the printed wiring board; alphabetic pins are on the bottom—or "foil"—side of the board.

Pin	*Designation*	*Function*
1	GND	Ground
2	PA3	Port-A bit 3
3	PA2	Port-A bit 2
4	PA1	Port-A bit 1
5	PA4	Port-A bit 4
6	PA5	Port-A bit 5
7	PA6	Port-A bit 6
8	PA7	Port-A bit 7
9	PB0	Port-B bit 0
10	PB1	Port-B bit 1
11	PB2	Port-B bit 2
12	PB3	Port-B bit 3
13	PB4	Port-B bit 4
14	PA0	Port-A bit 0
15	PB7	Port-B bit 7
16	PB5	Port-B bit 5
17	KB RO	Keyboard Row 0
18	KB CF	Keyboard Column F

Pin	*Designation*	*Function*
19	KB CB	Keyboard Column B
20	KB CE	Keyboard Column E
21	KB CA	Keyboard Column A
22	KB CD	Keyboard Column D
A	+5	+5-volt DC from main board power supply
B	K0	Memory-bank select signals (Active-LOW)
C	K1	
D	K2	
E	K3	
F	K4	
H	K5	
J	K7	
K	Decode	Memory decode signal used to increase memory capacity with off-board memory devices
L	AUD IN	Audio input from cassette
M	AUDOUTL	Low-level audio output to cassette with "micr" input
N	+12	+12-volt DC power from main board
P	AUDOUTH	High-level audio output to cassette player with "line" input
R	TTYKBD+	Positive terminal of 20-mA teletype keyboard loop
S	TTYPNT+	Positive terminal of 20-mA teletypewriter printer loop
T	TTYKBD−	Negative terminal of 20-mA teletypewriter keyboard loop
U	TTYPNT−	Negative terminal of 20-mA teletypewriter printer loop
V	KB R3	Keyboard Row 3
W	KB CG	Keyboard Column G
X	KB R2	Keyboard Row 2
Y	KB CC	Keyboard Column C
Z	KB R1	Keyboard Row 1

The KIM-1 and related computers use the 6522 VIA device. The 6522 contains two 8-bit I/O ports—Port-A and Port-B. These ports are represented by bits PA0–PA7 and PB0–PB7. Both ports can be configured under software control for either input or output port service on a bit-by-bit basis. In other words, PA0 might be an input bit, while PA1 is an output port bit. Or we can configure all 8 bits of either or both ports as either input or output.

KIM-1/SYM-1/AIM-65 Expansion Connector

Pin	*Designation*	*Function*
1	SYNC	Active-HIGH output line that goes HIGH during the phase-1 clock signal during instruction fetch operations. This line is used to allow the 6502 to operate with slow memory, dynamic memory, or in the Direct Memory Access mode.
2	$\overline{\text{RDY}}$	Has the effect of inserting a wait state into the CPU operating cycle. See similar description for same signal in Apple II discussion
3	ϕ_1	Phase-1 clock signal
4	$\overline{\text{IRQ}}$	Maskable interrupt request line. Active-LOW
5	RO	Reset overflow input. A negative-edge triggered input that will reset the overflow flip-flop in the CPU
6	$\overline{\text{NMI}}$	Active-LOW nonmaskable interrupt input line. This interrupt line cannot be masked by the internal interrupt flip-flop.
7	$\overline{\text{RST}}$	In parallel with the reset line on the 6502 and on the microcomputer. When brought LOW, this line will cause the program counter inside the 6502 to be loaded with 00H. The effect of this line is to form a hardware "JUMP to 00H" instruction.
8	DB7	Data Bus bit 7
9	DB6	Data Bus bit 6
10	DB5	Data Bus bit 5
11	DB4	Data Bus bit 4
12	DB3	Data Bus bit 3
13	DB2	Data Bus bit 2
14	DB1	Data Bus bit 1
15	DB0	Data Bus bit 0
16	K6	Address decoder output that goes HIGH whenever the COU addresses a location from 1800H to 1BFFH

16

6502 Detailed Instruction Set

The 6502 instruction set is presented in this chapter, so that you can study the instructions on a one-by-one basis. We will give you the common assembly language mnemonic for each instruction, a brief description to supplement the descriptions in Chapter 7, and the operations code (op-code) for each. The codes are listed in hexadecimal (HEX), binary, and octal formats to accommodate different computers.

ADC

Add Memory to Accumulator with Carry

A + M + C → A,C

Status Register Flags Affected: N, Z, C, V

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Immediate	ADC #oper	69	01101001	151	2	2
Zero Page	ADC oper	65	01100101	145	2	3
Zero Page,X	ADC oper,X	75	01110101	165	2	4
Absolute	ADC oper	6D	01101101	155	3	4*
Absolute,X	ADC oper,X	7D	01111101	175	3	4*
Absolute,Y	ADC oper,Y	79	01111001	171	3	6
(Indirect,X)	ADC (oper,X)	61	01100001	141	2	5*
(Indirect),Y	ADC (oper),Y	71	01110001	161	2	

*Add 1 if a page boundary is crossed.

The ADC instruction performs an addition with carry between the contents of the accumulator, the carry flag, and the contents of a memory location (specified or implied, depending upon the instruction). The status register flags are affected according to the following protocols:

Carry Flag (C). The carry flag is set (C = 1) if the sum of a binary addition exceeds FFH (255_{10}) or if the sum of a decimal (BCD) addition exceeds 99_{10}. All other results will cause the carry flag to be reset (C = 0).

Negative Flag (N). The negative flag will be set (N = 1) if bit 7 of the result stored in the accumulator is 1, and reset (N = 0) if bit 7 of the result is 0.

Overflow Flag (V). The overflow flag is set (V = 1) when the sign or bit 7 changes because the result in the accumulator is greater than $+127_{10}$ (7FH) or -128_{10}. All other results cause the overflow flag to be reset (V = 0).

Zero Flag (Z). The zero flag is set (Z = 1) if the result in the accumulator is 0, and reset (Z = 0) for all other results.

AND

Logical-AND Operation Between Memory and the Accumulator

$A \land M \rightarrow A$

Status Register Flags Affected: N, Z

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Immediate	AND #oper	29	00100001	51	2	2
Zero Page	AND oper	25	00100101	45	2	3
Zero Page,X	AND oper,X	35	00110101	65	2	4
Absolute	AND oper	2D	00100101	55	3	4
Absolute,X	AND oper,X	3D	00110101	75	3	4*
Absolute,Y	AND oper,Y	39	00110001	71	3	4*
(Indirect,X)	AND (oper,X)	21	00100001	41	2	6
(Indirect),Y	AND (oper),Y	31	00110001	61	2	5*

*Add 1 if page boundary is crossed.

The logical-AND instruction is of Group One, and has the full complement of addressing modes: Immediate, Absolute, Zero Page, Absolute X, Absolute Y, Zero Page X, Indexed Indirect, and Indirect Indexed.

The AND instruction causes the CPU to perform a logical-AND on a bit-by-bit basis between the contents of the accumulator and a data word fetched from memory. The results of the AND operation are stored in the accumulator. The rules for the logical-AND operation are:

$$\begin{aligned} 0 \text{ AND } 0 &= 0 \\ 0 \text{ AND } 1 &= 0 \\ 1 \text{ AND } 0 &= 0 \\ 1 \text{ AND } 1 &= 1 \end{aligned}$$

In the logical-AND instruction, the operation is on a bit-for-bit basis. The result of any operation on a given bit will not affect any other bit.

The Zero Flag (Z) is set (1) if the result in the accumulator is zero (00000000), otherwise it is reset (Z = 0). The Negative Flag (N) is set (N = 1) if accumulator bit 7 of the result is 1, and reset (N = 0) otherwise.

ASL

Shift Left 1-Bit Data in Either Accumulator or Memory

$$C \leftarrow \begin{array}{|c|c|c|c|c|c|c|c|} \hline 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ \hline \end{array} \leftarrow 0$$

Status Flags Affected: N, Z, C

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Accumulator	ASL A	0A	00001010	12	1	2
Zero Page	ASL oper	06	00000110	06	2	5
Zero Page,X	ASL oper,X	16	00010110	26	2	6
Absolute	ASL oper	0E	00001110	16	3	6
Absolute,X	ASL oper,X	1E	00011110	36	3	7

The ASL instruction will operate on either the accumulator or an addressed memory location. In the ASL instruction, bit 7 is always shifted to the carry flag (C), and bit 0 is made zero. The negative flag (N) will be made equal to the result in bit 7. The zero flag (Z) will be set (Z = 1) if the result is zero, and reset (Z = 0) otherwise. The carry flag contains the former bit 7 data (1 or 0).

BCC

Branch on Carry Clear (C = 0)

Status Register Flags Affected: None

Addressing Mode	*Mnemonic*	*Op-Code*			*No.*	*No.*
		Hex	*Binary*	*Octal*	*Bytes*	*Cycles*
Relative	BCC oper	90	10010000	220	2	2*

*Add 1 if branch occurs to same page, add 2 if branch occurs to another page.

This 2-byte instruction causes a relative branch forward or backward a number of steps specified by the second byte of the instruction code. Forward branches are specified by a positive hexadecimal number, while backward branches are represented by a two's complement equivalent hex negative number. For example, branching ahead 6 locations (+6) would be represented by 06H in the second byte, while branching 6 steps backwards (−6) is represented by FAH. The branch occurs if the Carry Flag is reset (C = 0).

BCS

Branch on Carry Set (C = 1)

Branch on C = 1

Status Register Flags Affected: None

Addressing Mode	*Mnemonic*	*Op-Code*			*No.*	*No.*
		Hex	*Binary*	*Octal*	*Bytes*	*Cycles*
Relative	BCS oper	B0	10110000	260	2	2*

*Add 1 if branch occurs to same page, add 2 if branch occurs to next page.

This 2-byte instruction is a relative branch forward or backward. The branch occurs a number of bytes of memory specified by the second byte of the instruction. Forward branches are specified by a hexadecimal number, while backward branches are represented by a two's complement equivalent. For example, branching ahead 6 locations would be represented by 06H in the second byte, while branching 6 steps backwards is represented by FAH (-6_{10}). Branch will occur when the carry flag of the Processor Status Register is set (C = 1).

BEQ

Branch on Result Equals Zero (Z = 1)

Branch on Z = 1

Status Register Affected: None

Addressing Mode	*Mnemonic*	*Op-Code Hex*	*Op-Code Binary*	*Op-Code Octal*	*No. Bytes*	*No. Cycles*
Relative	BEQ oper	F0	11110000	360	2	2*

*Add 2 if branch occurs to the same page, add 2 if to another page, i.e., if page boundary is crossed.

The BEQ instruction is a conditional branch that will branch when the result of an operation is zero (when the Z-flag is 1). If the result of an operation is zero, then the Z-flag is set (Z = 1); the BEQ instruction tests the Z-flag. If the Z-flag is 0, indicating a non-zero result, then no branch occurs and the program will execute the next instruction in sequence after BEQ.

Branching is relative, meaning that the program will jump forward or backward an amount specified by the second byte of the instruction. Forward jumps are specified by a positive hexadecimal number, while backward branches are specified by a two's complement hexadecimal equivalent number. For example, branching 6 steps forward would be specified by 06H, while 6 steps backward (−6) is represented by FAH. BEQ is the complement of the BNE (Branch on Not Equal) instruction.

BIT

Bit Test

Tests bits in memory with accumulator

$$A \wedge M,\ M_7 \rightarrow N,\ M_6 \rightarrow V$$

Status Register Flags Affected: N, Z, V

Addressing Mode	*Mnemonic*	*Op-Code*			*No.*	*No.*
		Hex	*Binary*	*Octal*	*Bytes*	*Cycles*
Zero Page	BIT oper	24	00100100	44	2	3
Absolute	BIT oper	2C	00101100	54	3	4

The BIT instruction is a comparison operation used to test bits from a memory location with bits in the accumulator. The contents of the accumulator are not affected by this instruction. Hence, the BIT instruction is termed "nondestructive."

The BIT instruction affects the N, V, and Z flags of the processor status register. The N-flag is set to the value of memory word bit 7 (M_7), while V is set to the value of memory word bit 6 (M_6). The Z-flag is set ($Z = 1$) if the result is zero, and reset ($Z = 0$) if the result is non-zero.

BIT performs a comparison by executing a logical-AND operation between the contents of the accumulator and the contents of a specified memory location. If the result of this operation is zero, then $Z = 1$, otherwise $Z = 0$.

BMI

Branch on Result Equals Minus (N = 1)

Branches when N = 1

Status Register Flags Affected: None

Addressing Mode	*Mnemonic*	*Op-Code*			*No.*	*No.*
		Hex	*Binary*	*Octal*	*Bytes*	*Cycles*
Relative	BMI oper	30	00110000	60	2	2*

*Add 1 if branch occurs to same page, add 2 if branch crosses page boundary.

The BMI instruction is a conditional branch instruction. The branch is taken if the result of a previous operation is negative, as indicated by the N-flag of the processor status register being set (N = 1). This test tells us that bit 7 of the previous result was 1.

The BMI instruction uses relative addressing. The branch occurs forward or backward a number of steps specified by the second byte of the instruction. A forward branch is denoted by a positive hexadecimal number, while a backward branch by an equivalent two's complement hexadecimal number. For example, a branch 6 spaces forward (+6) would be designed by 06H in the second byte, while a branch 6 steps backward (−6) by the hexadecimal equivalent FAH.

BNE

Branch on Result Not Equal to Zero (Z = 0)

Branches on Z = 0

Status Register Flags Affected: None

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Relative	BNE oper	D0	11010000	320	2	2*

*Add 1 if branch occurs to same page, add 2 if branch occurs to different page.

The BNE instruction is a conditional branch that takes the branch if the result of a previous instruction was not equal to zero. BNE tests the Z-flag of the processor status register, and will branch if Z = 0.

The BNE instruction uses relative addressing. The branch causes a jump forward or backward an amount specified in the second byte of the instruction. A forward branch is denoted by a positive hexadecimal number in the second byte, while a backward branch is indicated by a two's complement hexadecimal number. For example, a branch forward of 6 steps (+6) is denoted by 06H, while a branch backward of 6 spaces (−6) is denoted by FAH.

BPL

Branch on Result Positive

Branch on N = 0

Status Register Flags Affected: None

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Relative	BPL,oper	10	00010000	020	2	2*

*Add 1 if branch occurs to same page, add 2 if branch occurs to different page.

This instruction causes a branch when the result of the previous instruction was positive, as indicated by the N-flag being 0. Relative addressing mode is used, with the jump displacement being given by the second byte of the instruction. A positive jump will be indicated by a positive hexadecimal number, while a backward branch is indicated by a two's complement hexadecimal number in the second byte. For example, a forward branch of 6 steps is indicated by 06H in byte 2, while a backward branch (−6) is indicated by the hexadecimal number FAH.

BRK

Force Break

Forced interrupt PC + 2 ↓ P ↓

Status Register Flags Affected: I

Addressing Mode	*Mnemonic*	*Op-Code Hex*	*Op-Code Binary*	*Op-Code Octal*	*No. Bytes*	*No. Cycles*
Implied	BRK	18	00011000	030	1	7

The BRK command is a means for forcing the microprocessor to execute the interrupt subroutine under program control. The address of the first instruction of the interrupt subroutine is stored at locations FFFEH (low-order byte) and FFFFH (high-order byte). The contents of the program counter are incremented by 2 and then pushed onto the external stack during the execution of the interrupt subroutine. The BRK command is not disabled by the I-flag in the processor status register. The I-flag, which is an interrupt disable flag, is set HIGH (1) by the BRK instruction.

BVC

Branch on Overflow Clear

Branch on V = 0

Status Register Flags Affected: None

Addressing Mode	*Mnemonic*	*Op-Code Hex*	*Op-Code Binary*	*Op-Code Octal*	*No. Bytes*	*No. Cycles*
Relative	BVC oper	50	0101000	120	2	2*

*Add 1 if branch occurs to same page, add 2 if branch occurs to different page.

The BVC instruction causes a jump when the overflow flag (V) in the processor status register is clear (V = 0). Thus, BVC is a conditional branch instruction that uses relative addressing. The BVC instruction tests the V-flag of the processor, and will branch if V = 0. The branch will jump forward or backward an amount specified in the second byte of the instruction. A forward branch is denoted by a positive hexadecimal number in the second byte, while a backward branch is indicated by a two's complement hexadecimal number.

BVS

Branch on Carry Set

Branch on V = 1

Status Register Flags Affected: None

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Relative	BVS oper	70	01110000	160	2	2*

*Add 1 if branch occurs to same page, add 2 if branch occurs to different page.

This instruction is exactly like BVC, except that the branch occurs when the V-flag is set (V = 1).

CLC

Clear Carry Flag

$0 \rightarrow C$

Status Register Flags Affected: C goes to 0

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Implied	CLC	18	00011000	030	1	2

The CLC instruction causes the carry flag of the processor status register to become clear (C = 0).

CLD

Clear Decimal Mode

$0 \rightarrow D$

Status Register Flags Affected: D

Addressing Mode	*Mnemonic*	*Op-Code Hex*	*Op-Code Binary*	*Op-Code Octal*	*No. Bytes*	*No. Cycles*
Implied	CLD	D8	11011000	330	1	2

CLI

Clear Interrupt Disable Bit

$0 \rightarrow I$

Status Register Flags Affected: I

Addressing Mode	*Mnemonic*	*Op-Code Hex*	*Op-Code Binary*	*Op-Code Octal*	*No. Bytes*	*No. Cycles*
Implied	CLI	58	01011000	130	1	2

The CLI instruction clears the interrupt disable flag (also called the I-flag) in the 6502 CPU. Execution of this flag causes the I-flag to go to zero ($I = 0$). The implied addressing mode is used because there is only one possible destination, namely the I-flag of the processor status register. The purpose of the CLI instruction is to permit the 6502 to respond to interrupt requests from the outside world that are indicated by the $\overline{\text{IRQ}}$ line dropping LOW. The I-flag is normally set to $I = 1$ when the 6502 is first turned on and the $\overline{\text{RST}}$ line is activated. The programmer must insert a CLI instruction somewhere in the program before it is necessary to respond to maskable interrupts. This instruction and the companion SEI (set interrupt flag) can be used to turn the interrupt function on and off as needed.

CLV

Clear Overflow Flag

$0 \rightarrow V$

Status Register Flags Affected: V

Addressing Mode	*Mnemonic*	*Op-Code* *Hex*	*Binary*	*Octal*	*No. Bytes*	*No. Cycles*
Implied	CLV	B8	10111000	270	1	2

The CLV instruction is used to clear the overflow flag (also called the V-flag) of the processor status register to LOW (V = 0). Implied addressing is used since there is only one possible destination for the instruction. CLV is 1-byte instruction and affects no flags other than the V-flag.

CMP

Compare Memory with Accumulator

A − M

Status Register Flags Affected: N, Z, C

Addressing Mode	*Mnemonic*	*Op-Code* *Hex*	*Binary*	*Octal*	*No. Bytes*	*No. Cycles*
Immediate	CMP # Oper	C9	11001001	311	2	2
Zero Page	CMP Oper	C5	11000101	305	2	3
Zero Page,X	CMP Oper,X	D5	11010101	325	2	4
Absolute	CMP Oper	CD	11001101	315	3	4
Absolute,X	CMP Oper,X	DD	11011101	335	3	4*
Absolute,Y	CMP Oper,Y	D9	11011001	331	3	4*
(Indirect,X)	CMP (Oper,X)	C1	11000001	301	2	6
(Indirect,)Y	CMP (Oper),Y	D1	11010001	321	2	5*

*Add 1 if page boundary is crossed.

The compare (CMP) instruction compares data fetched from memory with data stored in the accumulator without altering the data in the accumulator. CMP can use all eight Group-I addressing modes, and three of the PSR flags: C, N, and Z. The use of the flags is different for this instruction than for others, and operates as follows:

1. *C-flag.* Set HIGH (1) when the value in memory is less than or equal to the value in the accumulator, and is reset LOW (0) when the value in memory is greater than the value in the accumulator.

2. *N-flag* is set HIGH (1) or reset LOW (0) according to the result of bit 7.
3. *Z-flag* is set HIGH (1) on equal comparison, reset for unequal comparison.

CPX

Compare Memory with Index X-Register

X − M

Status Register Flags Affected: N, Z, C

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Immediate	CPX #oper	E0	11100000	340	2	2
Zero Page	CPX oper	E4	11100100	344	2	3
Absolute	CPX oper	EC	11101100	354	3	4

The CPX instruction compares the contents of the X-register with the contents of a designated memory location. Immediate, zero page, and Absolute addressing modes are used. The N, Z, and C-flags are affected. The contents of the X-register are not affected by CPX. The comparison is performed by subtracting the contents of the addressed memory location from the contents of the X-register, but does not store the result in either the X-register or the memory location. The PSR flags are affected as follows:

1. The C-flag will be set (C = 1) if the absolute value of the X-register is equal to or greater than the value fetched from memory (X M). The C-flag is reset (C = 0) if X is less than the value from memory.
2. If bit 7 of the comparison result is 1, then the N-flag is set (N = 1), but if bit 7 is 0, then the N-flag is reset (N = 0).
3. The Z-flag is set (Z = 0) if the value memory is equal to the value from the X-register, otherwise it is reset (Z = 0).

The CPX instruction can be used for setting of the PSR flags, among other uses.

CPY

Compare Memory with Index Y-Register

Y − M

Status Register Flags Affected: N, Z, C

Addressing Mode	*Mnemonic*	*Op-Code Hex*	*Op-Code Binary*	*Op-Code Octal*	*No. Bytes*	*No. Cycles*
Immediate	CPY #oper	C0	11000000	300	2	2
Zero Page	CPY oper	C4	11000100	304	2	3
Absolute	CPY oper	CC	11001100	314	4	4

The CPY instruction is exactly like the CPX instruction, except that the Y-register is used instead of the X-register. For a detailed discussion of this instruction, read the text for the CPX instruction, substituting "Y" for "X."

DEC

Decrement Memory by One

M − 1 → M

Status Register Flags Affected: N, Z

Addressing Mode	*Mnemonic*	*Op-Code Hex*	*Op-Code Binary*	*Op-Code Octal*	*No. Bytes*	*No. Cycles*
Zero Page	DEC oper	C6	11000110	306	2	5
Zero Page,X	DEC oper,X	D6	11010110	326	2	6
Absolute	DEC oper	CE	11001110	316	3	6
Absolute,X	DEC oper,X	DE	11011110	336	3	0

The DEC instruction causes the data in the addressed memory location to be decremented, i.e., decreased by one; the DEC instruction does not affect the accumulator data. The N and Z-flags of the processor status register are affected as follows:

1. The N-flag will be 1 when bit 7 of the result is 1, and 0 when the result is 0.
2. The Z-flag will be 1 when the result is zero (00000000), and 0 when the result is anything other than 00000000.

DEX

Decrement Index X-Register by One

$X - 1 \rightarrow X$

Status Register Flags Affected: N, Z

Addressing Mode	*Mnemonic*	*Op-Code Hex*	*Op-Code Binary*	*Op-Code Octal*	*No. Bytes*	*No. Cycles*
Implied	DEX	CA	11001010	312	1	2

The DEX instruction causes the data in the index X-register to be decremented by 1; the DEX instruction does not affect the contents of the accumulator or any memory location. The N and Z-flags of the processor status register are affected as follows:

1. The N-flag will be 1 when bit 7 of the result in the X-register is 1, and 0 when the result bit 7 is 0.
2. The Z-flag will be 1 when the result in the X-register is zero (00000000) and 0 when the result in the X-register is anything other than 00000000.

DEY

Decrement Index Y-Register by One

$Y - 1 \rightarrow Y$

Status Register Flags Affected: N, Z

Addressing Mode	*Mnemonic*	*Op-Code Hex*	*Op-Code Binary*	*Op-Code Octal*	*No. Bytes*	*No. Cycles*
Implied	DEY	88	10001000	210	1	2

The DEY instruction is identical in all respects to the DEX instruction, except that the index Y-register is used instead of index X-register.

EOR

Exclusive-OR (Logical Operation) Memory with Accumulator

$A \veebar M \rightarrow A$

Status Register Flags Affected: N, Z

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Immediate	EOR #oper	49	01001001	111	2	2
Zero Page	EOR oper	45	01000101	105	2	3
Zero Page,X	EOR oper,X	55	01010101	125	2	4
Absolute	EOR oper	4D	01001101	115	3	4
Absolute,X	EOR oper,X	5D	01011101	135	3	4*
Absolute,Y	EOR oper,Y	59	01011001	131	3	4*
(Indirect,X)	EOR (oper,X)	41	01000001	101	2	6
(Indirect),Y	EOR (oper),Y	51	01010001	121	2	5*

*Add 1 if page boundary is crossed.

The EOR instruction causes an exclusive-OR logical operation between the contents of the accumulator and the contents of the addressed memory location. The operation takes place on a bit-by-bit basis, so the result of one operation will not affect the operation on the next bit. The rules for EOR are as follows:

0 XOR 0 = 0

0 XOR 1 = 1

1 XOR 0 = 1

1 XOR 1 = 0

Note that the result bit is true (1) if either bit is true, but not if both bits are true. The EOR instruction affects the N and Z-flags of the processor status register as follows:

1. The N-flag will be 1 if bit 7 of the result is 1, and 0 if bit 7 of the result is 0.
2. The Z-flag will be 1 if the result of the operation stored in the accumulator is 00000000, and 1 if the result stored in the accumulator is anything other than 00000000.

One application for the EOR instruction is in complementing the accumulator, i.e., making all the 1s into 0s and all the 0s into 1s.

INC

Increment Memory by One

M + 1 → M

Status Register Flags Affected: N, Z

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Zero Page	INC oper	E6	11100110	346	2	5
Zero Page,X	INC oper,X	F6	11110110	366	2	6
Absolute	INC oper	EE	11101110	356	3	6
Absolute,X	INC oper,X	FE	11111110	376	3	7

The INC instruction causes the data in the addressed memory location to be incremented (increased) by one; the data in the accumulator is not affected by INC. The N and Z-flags of the processor status register are affected as follows:

1. The N-flag will be 1 if bit 7 of the result is 1, and 0 if bit 7 of the result is 0.
2. The Z-flag will be 1 if the result is 00000000, and 0 if the result is anything other than 00000000.

INX

Increment Index X-Register by One

X + 1 → X

Status Register Flags Affected: N, Z

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Implied	INX	E8	11101000	350	1	2

The INX instruction causes the data stored in the index X-register to be incremented (increased) by one; the accumulator data is not affected by INX. The N and Z-flags of the processor status register are affected as follows:

1. The N-flag will be 1 if bit 7 of the result stored in the X-register is 1, and 0 if bit 7 of the result is 0.
2. The Z-flag will be 1 if the result stored in the X-register is 00000000, and 0 if the result in the X-register is anything other than 00000000.

INY

Increment Index Y-Register by One

$Y + 1 \rightarrow Y$

Status Register Flags Affected: N, Z

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Implied	INY	C8	11001000	310	1	2

JMP

Jump to a New Location in Memory

$(PC + 1) \rightarrow PCL$

$(PC + 2) \rightarrow PCH$

Status Register Flags Affected: None

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Absolute	JMP oper	4C	01001100	114	3	3
Indirect	JMP (oper)	6C	01101100	154	3	3

The JMP (jump) instruction causes an immediate, unconditional transfer of program control to another memory location. Both Absolute and indirect addressing modes can be used. The next instruction to be executed after the JMP instruction will not be the next instruction in sequence (except for the trivial case where someone gratuitously added a "JMP to next location" instruction), but rather the instruction at the memory location specified by the operand of the instruction. Consider the following example:

Step	*Location*	*Instruction*	*Comment*
1	0200H	JMP nnnn	JUMP to location A008
2	0201H	08H	(nn)
3	0202H	A0H	(nn)
4	0203H	NOP	Next instruction in sequence

In step 1, the 6502 encounters a 3-byte JMP A008H instruction (step 2 is the low-order destination byte, while step 3 is the high-order destination byte—the two together make up the 16-bit address). The next instruction to be executed will not be the NOP (no operation) found at location 0203H in step 4, but rather it will be the instruction found at location A008H.

JSR

Jump to New Location for Subroutine (With Return Address)

PC + 2 ↓

(PC + 1) → PCL

(PC + 2) → PCH

Status Register Flags Affected: None

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Absolute	JSR oper	20	00100000	40	3	6

The JSR is a jump instruction similar to the JMP instruction, with the exception that JSR will store the 2 bytes of the last instruction address to be executed on the external stack (usually in page one of memory), and will then decrement the Stack Pointer (SP) register by 2. When the program returns from the subroutine (which it does on encountering an RTS instruction), the program counter will be loaded with the address of the next instruction to be executed after the subroutine is completed (i.e., the next instruction in sequence after JSR). On return from the subroutine, the following status exists:

PCL + 1 → PCL

PCH + 2 → PCH

The RTS (return from subroutine) must be the last instruction in the subroutine, otherwise the program control will not return to the main program.

LDA

Load Accumulator with Data Stored in Memory

M → A

Status Register Flags Affected: N, Z

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Immediate	LDA #oper	A9	10101001	251	2	2
Zero Page	LDA oper	A5	10100101	245	2	3
Zero Page,X	LDA oper,X	B5	10110101	265	2	4
Absolute	LDA oper	AD	10101101	255	3	4
Absolute,X	LDA oper,X	BD	10111101	275	3	4*
Absolute,Y	LDA oper,Y	B9	10111001	271	3	4*
(Indirect,X)	LDA (oper,X)	A1	10100001	241	2	6
(Indirect),Y	LDA (oper),Y	B1	10110001	261	2	5*

*Add 1 if page boundary is crossed.

The LDA instruction serves to load the accumulator with data taken from defined memory locations. This transfer of data is nondestructive, i.e., the data will appear both in the accumulator and in the original memory location when the instruction is executed. Thus, LDA is a copying operation rather than a transfer in the strict sense of the word. The LDA instruction uses all 8 addressing modes available to Group-I instructions; the operation of LDA with respect to these modes is described in Chapter 7. Only the N and Z-flags of the processor status register are affected by the LDA instruction:

1. The N-flag will be 1 if the data transferred into the accumulator has bit 7 = 1, and 0 if bit 7 = 0.
2. The Z-flag will be 1 if the data transferred into the accumulator is 00000000, and 1 if the data transferred into the accumulator is any number other than 00000000.

LDX

Load Index X-Register with Data Stored in Memory

M → X

Status Register Flags Affected: N, Z

Addressing Mode	*Mnemonic*	*Op-Code*			*No. Bytes*	*No. Cycles*
		Hex	*Binary*	*Octal*		
Immediate	LDX #oper	A0	10100000	240	2	2
Zero Page	LDX oper	A4	10100100	244	2	3
Zero Page,X	LDX oper,X	B4	10110100	264	2	4
Absolute	LDX oper	AC	10101100	254	3	4*
Absolute,X	LDX oper,X	BC	10111100	274	3	4*

*Add 1 if page boundary is crossed.

The LDX instruction loads the index X-register with data fetched from a defined memory location. The transfer of data is nondestructive, i.e., the data will appear in both the accumulator of the 6502 and in the original memory location after the execution of LDX. Thus, LDX is a copying operation rather than a transfer in the strict sense of the word. The LDX instruction uses only 5 of the 8 addressing modes available on 6502. The N and Z-flags of the processor status register are affected as follows:

1. The N-flag will be 1 if the data transferred into the X-register has bit 7 = 1, and 0 if bit 7 = 0.
2. The Z-flag will be 1 if the data transferred into the X-register is 00000000, and 0 if the data transferred is anything other than 00000000.

LDY

Load Index Y-Register with Data Stored in Memory

M → Y

Status Register Flags Affected: N, Z

Addressing Mode	*Mnemonic*	*Op-Code*			*No. Bytes*	*No. Cycles*
		Hex	*Binary*	*Octal*		
Immediate	LDY #oper	A0	10100000	240	2	2
Zero Page	LDY oper	A4	10100100	244	2	3
Zero Page,X	LDY oper,X	B4	10110100	264	2	4
Absolute	LDY oper	AC	10101100	254	3	4
Absolute,X	LDY oper,X	BC	10111100	274	3	4*

*Add 1 if page boundary is crossed.

The LDY instruction operates in exactly the same manner as the LDX instruction, except that the Y-register is the destination rather than the X-register (see the discussion for LDX).

LSR

Shift Right One Bit (Memory or Accumulator)

$0 \rightarrow b7 \rightarrow b6 \rightarrow b5 \rightarrow b4 \rightarrow b3 \rightarrow b2 \rightarrow b1 \rightarrow b0 \quad C$

Status Register Flags Affected: N, Z, C (Note: *N* goes to 0)

Addressing Mode	*Mnemonic*	*Op-Code*			*No. Bytes*	*No. Cycles*
		Hex	*Binary*	*Octal*		
Accumulator	LSR A	4A	01001010	112	1	2
Zero Page	LSR oper	46	01000110	106	2	5
Zero Page,X	LSR oper,X	56	01010110	126	2	6
Absolute	LSR oper	4E	01001110	116	3	6
Absolute,X	oper,X	5E	01011110	136	3	7

The LSR, or *logical shift right* instruction, causes a 0 to be shifted into bit 7 (also the N-flag of the processor status register) of the accumulator or memory location addressed; the contents of each bit is moved one position to the right, and bit 0 (the LSB) is moved to the carry flag. The LSR instruction uses the following addressing modes: accumulator, zero page, zero page X, absolute, and absolute X. See Chapter 7 for a discussion of the LSR (and the companion ASL instruction) and possible applications.

NOP

No Operation

Status Register Flags Affected: None

Addressing Mode	*Mnemonic*	*Op-Code* *Hex*	*Binary*	*Octal*	*No. Bytes*	*No. Cycles*
Implied	NOP	EA	11101010	352	1	2

The NOP instruction carries out no operation in the 6502 and affects no processor status register flags. The purpose of the NOP instruction is to insert a 2-cycle "wait" into a program or to take up a space.

ORA

Logical-OR Operation Between Accumulator and Memory Location

$A \vee M \rightarrow A$

Status Register Flags Affected: N, Z

Addressing Mode	*Mnemonic*	*Op-Code* *Hex*	*Binary*	*Octal*	*No. Bytes*	*No. Cycles*
Immediate	ORA #oper	09	00001001	011	2	2
Zero Page	ORA oper	05	00000101	005	2	3
Zero Page,X	ORA oper,X	15	00010101	025	2	4
Absolute	ORA oper	0D	00001101	015	3	4
Absolute,X	ORA oper,X	1D	00011101	035	3	4
Absolute,Y	ORA oper,Y	19	00011001	031	3	4
(Indirect,X)	ORA (oper,X)	01	00000001	001	2	6
(Indirect),Y	ORA (oper),Y	11	00010001	021	2	5*

*Add 1 if page boundary is crossed.

The ORA instruction performs a logical-OR operation on a bit-by-bit basis between the accumulator and the addressed memory location. The operation affects the contents of the accumulator. The rules for a logical-OR operation are as follows:

0 OR 0 = 0

0 OR 1 = 1

1 OR 0 = 1

1 OR 1 = 1

As you can see from this, the result of a logical-OR operation is true (1) any time either of the bits being compared is 1. Since the ORA instruction operates on a bit-for-bit basis, no operation between bits of any order will affect the operation of the ORA instruction on any other set of bits. The ORA instruction affects the N and Z-flags of the processor status register as follows:

1. The N-flag will be 1 if bit 7 of the result in the accumulator is 1, and 0 if bit 7 is 0.
2. The Z-flag will be 1 if the result in the accumulator is 00000000, and 0 if the result is anything other than 00000000.

PHA

Push Accumulator Contents Onto External Stack

A ↓

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Implied	PHA	48	01001000	110	1	3

The PHA instruction serves to push the contents of the accumulator out onto the external stack in memory. The PHA instruction doesn't affect any processor status register flags, but will cause the Stack Point (SP) to decrement by one.

PHP

Push Processor Status Register Onto External Stack

P ↓

Status Register Flags Affected: None

Addressing		*Op-Code*			*No.*	*No.*
Mode	*Mnemonic*	*Hex*	*Binary*	*Octal*	*Bytes*	*Cycles*
Implied	PHP	08	00001000	010	1	3

The PHP instruction is exactly like PHA, except that the contents of the processor status register are transferred to the external stack, rather than the accumulator contents. No PSR flags are affected, but the instruction does cause the SP to decrement by 1.

PLA

Pull Accumulator From Stack

A ↑

Status Register Flags Affected: N, Z

Addressing Mode	*Mnemonic*	*Op-Code*			*No.*	*No.*
		Hex	*Binary*	*Octal*	*Bytes*	*Cycles*
Implied	PLA	68	01101000	150	1	4

The PLA instruction is used to pull data from the external stack back to the accumulator. This instruction is thus the opposite of the PHA instruction. The N and Z-flags of the processor status register are affected by this operation as follows:

1. The N-flag will be 1 if the returned data has bit 7 = 1, and 0 if bit 7 is 0.
2. The Z-flag will be 1 if the returned data is 00000000, and 0 if the returned data is anything other than 00000000.

PLP

Pull Processor Status From External Stack

P ↑

Status Flags Affected: All

Addressing Mode	*Mnemonic*	*Op-Code*			*No.*	*No.*
		Hex	*Binary*	*Octal*	*Bytes*	*Cycles*
Implied	PLP	28	00101000	050	1	4

The PLP instruction works exactly like the PLA instruction, except that the data pulled from the stack are stored in the processor status register instead of the accumulator. All of the PSR flags are affected, and become whatever the corresponding bits were on the external stack. One use of this instruction is to restore the PSR after some alternative operation, such as a subroutine.

ROL

Rotate Data One Bit to the Left (Memory Location or Accumulator)

$$\underbrace{b7 \leftarrow b6 \leftarrow b5 \leftarrow b4 \leftarrow b3 \leftarrow b2 \leftarrow b1 \leftarrow b0}_{\text{(M or A)}} \leftarrow \text{C} \leftarrow$$

Status Register Flags Affected: N, Z, C

Addressing Mode	*Mnemonic*	*Op-Code* *Hex*	*Binary*	*Octal*	*No. Bytes*	*No. Cycles*
Accumulator	ROL A	2A	00101010	052	1	2
Zero Page	ROL oper	26	00100110	046	2	5
Zero Page,X	ROL oper,X	36	00110110	066	2	6
Absolute	ROL oper	2E	00101110	056	3	6
Absolute,X	ROL oper,X	3E	00111110	076	3	7

The ROL instruction is like the ASL instruction, except that the shifted-out data recirculates, i.e., the bit 7 datum is stored in the C-flag, while the C-flag data is stored in B0. Either the accumulator or a byte from memory can be handled with ROL. Each bit of the affected byte is shifted 1 place to the left, as shown in the diagram. The N, Z, and C-flags of the processor status register are affected as follows:

1. The N-flag will be 1 if bit 7 of the result is 1, and 0 if bit 7 is 0.
2. The Z-flag will be 1 if the result is 00000000, and 0 if the result is anything other than 00000000.
3. The C-flag takes on the value (1 or 0) that was in bit 7 before the shift occurred.

ROR

Rotate One Bit to the Right (Memory or Accumulator)

$$\rightarrow \underbrace{b7 \rightarrow b6 \rightarrow b5 \rightarrow b4 \rightarrow b3 \rightarrow b2 \rightarrow b1 \rightarrow b0}_{\text{(M or A)}} \rightarrow \text{C}$$

Status Register Flags Affected: N, Z, C

Addressing Mode	*Mnemonic*	*Op-Code*			*No.*	*No.*
		Hex	*Binary*	*Octal*	*Bytes*	*Cycles*
Accumulator	ROR A	6A	01101010	152	1	2
Zero Page	ROR oper	66	01100110	146	2	5
Zero Page,X	ROR oper,X	76	01110110	166	2	6
Absolute	ROR oper	6E	01101110	156	3	6
Absolute,X	ROR oper,X	7E	01111110	176	3	7

The ROR instruction is similar to the LSR instruction, except that the shifted-out data is recirculated back into the register. Bit 0 data are right-shifted into the C-flag, while the previous contents of the C-flag are shifted into bit 7. All other bits are shifted 1 place to the right, as shown in the diagram above. The C, Z, and N-flags of the processor status register are affected in the same manner as for the ROL instruction.

RTI

Return From Interrupt

P ↑

PC ↑

Status Register Flags Affected: All

Addressing Mode	*Mnemonic*	*Op-Code*			*No.*	*No.*
		Hex	*Binary*	*Octal*	*Bytes*	*Cycles*
Implied	RTI	40	01000000	100	1	6

The RTI instruction allows the 6502 microprocessor to return from serving an interrupt. When the 6502 encounters the RTI instruction, it will restore the previous program by pulling the previous processor status register (P) and program counter contents from memory. Thus, the processor status register will return to its condition when the interrupt was encountered, as will the PC. Since no other registers are affected, the programmer may wish to save the contents of other registers (if they are of importance) with other steps in the program. RTI must be the last instruction in the interrupt service subroutine.

RTS

Return From Subroutine

PC ↑

PC + 1 → PC

Status Register Flags Affected: None

Addressing Mode	*Mnemonic*	*Op-Code* *Hex*	*Binary*	*Octal*	*No. Bytes*	*No. Cycles*
Implied	RTS	60	01100000	140	1	6

The RTS instruction performs for subroutines (see instructions for JSR) what RTI does for interrupt service routines. The purpose of the RTS instruction is to restore the 6502 processor to the program being executed when the subroutine instruction JSR was encountered. The program counter is returned from the external stack and then incremented by 1, the new value being stored in the PC of the 6502 to point to the next instruction in sequence after the JSR was encountered. The RTS instruction must be the last instruction in a subroutine program, otherwise, the processor will not know how to return to the main program.

SBC

Subtract Memory From Accumulator with Borrow

$A - M - \overline{C} \rightarrow A$ (Note: "$\overline{C}$" denotes a borrow operation)

Status Register Flags Affected: N, Z, Z, V

Addressing Mode	*Mnemonic*	*Op-Code* *Hex*	*Binary*	*Octal*	*No. Bytes*	*No. Cycles*
Immediate	SBC	E9	11101001	351	2	2
Zero Page	SBC	E5	11100101	345	2	3
Zero Page,X	SBC	F5	11110101	365	2	4
Absolute	SBC	ED	11101101	355	3	4
Absolute,X	SBC	FD	11111101	375	3	4*
Absolute,Y	SBC	F9	11111001	371	3	4*
(Indirect,X)	SBC	E1	11100001	341	2	6
(Indirect),Y	SBC	F1	11110001	361	2	5*

*Add 1 if page boundary is crossed.

The SBC (subtraction with carry) instruction is actually a subtraction with BORROW, if we use mathematically correct terminology. The symbolic operation for SBC is

$$A - M - \overline{C} \rightarrow A$$

This notation says that the value fetched from memory (M) and the complement of the carry flag ($\overline{C}$) is subtracted from the contents of the accumulator, and the result is stored in the accumulator. Note that the carry flag will be set (HIGH) if a result is equal to or greater than zero, and reset (LOW) if the results are less than zero, i.e., negative.

The SBC instruction has available all 8 Group-I addressing modes, as was also true of ADC.

The SBC instruction affects the following PSR flags: negative (N), zero (Z), Carry (C), and overflow (V). The N-flag indicates a negative result and will be HIGH; the Z-flag is HIGH if the result of the SBC instruction is zero and LOW otherwise; the overflow flag (V) is HIGH when the result exceeds the values 7FH ($+127_{10}$) and 80H with C = 1 (i.e., -128_{10}).

The 6502 manufacturer recommends for single-precision (8-bit) subtracts that the programmer ensure that the carry flag is set prior to the SBC operation to be sure that true two's complement arithmetic takes place. We can set the carry flag by executing the SEC (set carry flag) instruction.

The rules for binary subtraction are:

$$\begin{array}{ll} 0 - 0 = 0 & \\ 0 - 1 = 0 & \text{Carry} - 1 \\ 1 - 0 = 1 & \\ 1 - 1 = 0 & \end{array}$$

The SBC instruction complements the ADC instruction and is used in arithmetic operations. The additional instruction used in arithmetic operations is the set decimal mode instruction that permits binary coded decimal (BCD) arithmetic.

SEC

Set Carry Flag

$1 \rightarrow C$

Status Register Flags Affected: C goes to 1

Addressing Mode	*Mnemonic*	*Op-Code Hex*	*Op-Code Binary*	*Op-Code Octal*	*No. Bytes*	*No. Cycles*
Implied	SEC	38	00111000	070	1	2

Sets processor status register C-flag to 1.

SED

Set Decimal Mode

1 → D

Status Flags Affected: D goes to 1

Addressing Mode	*Mnemonic*	*Op-Code Hex*	*Op-Code Binary*	*Op-Code Octal*	*No. Bytes*	*No. Cycles*
Implied	SED	F8	11111000	370	1	2

Sets processor status register D-flag to 1, thereby permitting decimal operations (see Chapter 7).

SEI

Set Interrupt Disable Status Bit

1 → I

Addressing Mode	*Mnemonic*	*Op-Code Hex*	*Op-Code Binary*	*Op-Code Octal*	*No. Bytes*	*No. Cycles*
Implied	SEI	78	01111000	170	1	2

Sets interrupt status bit (I-flag) of the processor status register to 1, thereby disabling the interrupt capability of the 6502.

STA

Store Accumulator Contents in Memory

A → M

Status Register Flags Affected: None

Addressing Mode	*Mnemonic*	*Op-Code*			*No.*	*No.*
		Hex	*Binary*	*Octal*	*Bytes*	*Cycles*
Zero Page	STA oper	85	10000101	205	2	3
Zero Page,X	STA oper,X	95	10010101	225	2	4
Absolute	STA oper	8D	10001101	215	3	4
Absolute,X	STA oper,X	9D	10011101	235	3	5
Absolute,Y	STA oper,Y	99	10011001	231	3	5
(Indirect,X)	STA (oper,X)	81	10000001	201	2	6
(Indirect),Y	STA (oper),Y	91	10010001	221	2	6

The STA instruction stores the contents of the accumulator in a location in memory. Seven modes of addressing are available, as detailed here. The transfer is nondestructive, so the same data will appear in both the accumulator and in the selected memory location immediately after the execution of an STA instruction.

STX

Store Index X-Register in Memory

X → M

Status Register Flags Affected: None

Addressing Mode	*Mnemonic*	*Op-Code*			*No.*	*No.*
		Hex	*Binary*	*Octal*	*Bytes*	*Cycles*
Zero Page	STX oper	86	10000110	206	2	3
Zero Page,Y	STX oper,Y	96	10010110	226	2	4
Absolute	STX oper	8E	10001110	216	3	4

The STX instruction stores the contents of the index X-register in a location in memory. Three different addressing modes are allowed: zero page, zero page-Y, and Absolute. The transfer is nondestructive, so the same data will appear in both the X-register and the selected memory location following the transfer.

STY

Store Index Y-Register in Memory

Y → M

Addressing Mode	*Mnemonic*	*Op-Code*			*No. Bytes*	*No. Cycles*
		Hex	*Binary*	*Octal*		
Zero Page	STY oper	84	10000100	204	2	3
Zero Page,X	STY oper,X	94	10010100	224	2	4
Absolute	STY oper	8C	10001100	214	3	4

The STY instruction stores the contents of the index Y-register in a location in memory. Three different addressing modes are allowed: zero page, zero page-X, and Absolute. The transfer is nondestructive, so the same data will appear in both the Y-register and in the selected memory location immediately after execution of the STY instruction.

TAX

Transfer Contents of Accumulator to Index X-Register

A → X

Status Register Flags Affected: N, Z

Addressing Mode	*Mnemonic*	*Op-Code*			*No. Bytes*	*No. Cycles*
		Hex	*Binary*	*Octal*		
Implied	TAX	AA	10101010	252	1	2

The TAX instruction transfers the data in the accumulator into the index X-register. This transfer is nondestructive, so the same data will appear in both the accumulator and the X-register following execution of this instruction. The N and Z-flags of the processor status register are affected as follows:

1. The N-flag will be 1 if bit 7 of the data transferred into the X-register is 1, and 0 if bit 7 of the transferred data is 0.
2. The Z-flag will be 1 if the data transferred to the X-register is 00000000, and 0 if the transferred data is anything other than 00000000.

TYA

Transfer Contents of Index Y-Register to the Accumulator

Y → A

Status Register Flags Affected: N, Z

Addressing Mode	*Mnemonic*	*Op-Code*			*No.*	*No.*
		Hex	*Binary*	*Octal*	*Bytes*	*Cycles*
Implied	TYA	98	10011000	230	1	2

This instruction is the same as TAX, except that data is transferred from the index Y-register to the accumulator. The treatment of the flags is the same.

TSX

Transfer Stack Pointer to Index X-Register

$S \rightarrow X$

Status Register Flags Affected: N, Z

Addressing Mode	*Mnemonic*	*Op-Code*			*No.*	*No.*
		Hex	*Binary*	*Octal*	*Bytes*	*Cycles*
Implied	TSX	BA	10111010	272	1	2

Transfers contents of the stack pointer (SP) to the X-register. The N and Z-flags are affected in the same manner as for TAX.

TXA

Transfer Index X-Register to Accumulator

$X \rightarrow A$

Status Register Flags Affected: N, Z

Addressing Mode	*Mnemonic*	*Op-Code*			*No.*	*No.*
		Hex	*Binary*	*Octal*	*Bytes*	*Cycles*
Implied	TXA	8A	10001010	212	1	2

The TXA instruction transfers the contents of the X-register into the accumulator. The TXA instruction is opposite of the TAX instruction, and affects the N and Z-flags in exactly the same manner.

TXS

Transfer Index X-Register to Stack Pointer

X → SP

Status Register Flags Affected: None

Addressing Mode	*Mnemonic*	*Op-Code Hex*	*Op-Code Binary*	*Op-Code Octal*	*No. Bytes*	*No. Cycles*
Implied	TXS	9A	10011010	232	1	2

This instruction transfers the contents of the index X-register into the accumulator.

APPENDICES

APPENDIX A

INSTRUCTION ADDRESSING MODES AND RELATED EXECUTION TIMES (in clock cycles)

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect, X)	(Indirect), Y	Absolute Indirect
ADC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
AND	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
ASL	2	.	5	6	.	6	7	.	.	.	.	.	.
BCC	.	.	.	.	.	.	.	.	.	2**	.	.	.
BCS	.	.	.	.	.	.	.	.	.	2**	.	.	.
BEQ	.	.	.	.	.	.	.	.	.	2**	.	.	.
BIT	.	.	3	.	.	4	.	.	.	.	.	.	.
BMI	.	.	.	.	.	.	.	.	.	2**	.	.	.
BNE	.	.	.	.	.	.	.	.	.	2**	.	.	.
BPL	.	.	.	.	.	.	.	.	.	2**	.	.	.
BRK	.	.	.	.	.	.	.	.	.	.	.	.	.
BVC	.	.	.	.	.	.	.	.	.	2**	.	.	.
BVS	.	.	.	.	.	.	.	.	.	2**	.	.	.
CLC	.	.	.	.	.	.	.	.	2	.	.	.	.
CLD	.	.	.	.	.	.	.	.	2	.	.	.	.
CLI	.	.	.	.	.	.	.	.	2	.	.	.	.
CLV	.	.	.	.	.	.	.	.	2	.	.	.	.
CMP	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
CPX	.	2	3	.	.	4	.	.	.	.	.	.	.
CPY	.	2	3	.	.	4	.	.	.	.	.	.	.
DEC	.	.	5	6	.	6	7	.	.	.	.	.	.
DEX	.	.	.	.	.	.	.	.	2	.	.	.	.
DEY	.	.	.	.	.	.	.	.	2	.	.	.	.
EOR	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
INC	.	.	5	6	.	6	7	.	.	.	.	.	.
INX	.	.	.	.	.	.	.	.	2	.	.	.	.
INY	.	.	.	.	.	.	.	.	2	.	.	.	.
JMP	.	.	.	.	.	3	.	.	.	.	.	.	5

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect, X)	(Indirect),Y	Absolute Indirect
JSR	.	.	.	.	.	6	.	.	.	.	.	.	.
LDA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
LDX	.	2	3	.	4	4	.	4*	.	.	.	.	.
LDY	.	2	3	4	.	4	4*	.	.	.	.	.	.
LSR	2	.	5	6	.	6	7	.	.	.	.	.	.
NOP	.	.	.	.	.	.	.	.	2	.	.	.	.
ORA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
PHA	.	.	.	.	.	.	.	.	3	.	.	.	.
PHP	.	.	.	.	.	.	.	.	3	.	.	.	.
PLA	.	.	.	.	.	.	.	.	4	.	.	.	.
PLP	.	.	.	.	.	.	.	.	4	.	.	.	.
ROL	2	.	5	6	.	6	7	.	.	.	.	.	.
ROR	2	.	5	6	.	6	7	.	.	.	.	.	.
RTI	.	.	.	.	.	.	.	.	6	.	.	.	.
RTS	.	.	.	.	.	.	.	.	6	.	.	.	.
SBC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
SEC	.	.	.	.	.	.	.	.	2	.	.	.	.
SED	.	.	.	.	.	.	.	.	2	.	.	.	.
SEI	.	.	.	.	.	.	.	.	2	.	.	.	.
STA	.	.	3	4	.	4	5	5	.	.	6	6	.
STX	.	.	3	.	4	4	.	.	.	.	.	.	.
STY	.	.	3	4	.	4	.	.	.	.	.	.	.
TAX	.	.	.	.	.	.	.	.	2	.	.	.	.
TAY	.	.	.	.	.	.	.	.	2	.	.	.	.
TSX	.	.	.	.	.	.	.	.	2	.	.	.	.
TXA	.	.	.	.	.	.	.	.	2	.	.	.	.
TXS	.	.	.	.	.	.	.	.	2	.	.	.	.
TYA	.	.	.	.	.	.	.	.	2	.	.	.	.

* Add one cycle if indexing across page boundary
** Add one cycle if branch is taken, Add one additional if branching operation crosses page boundary

APPENDIX B 6502 Instructions Sorted by Op-Code (Hexadecimal)

ØØ	- BRK
Ø1	- ORA - (Indirect,X)
Ø2	- Future Expansion
Ø3	- Future Expansion
Ø4	- Future Expansion
Ø5	- ORA - Zero Page
Ø6	- ASL - Zero Page
Ø7	- Future Expansion
Ø8	- PHP
Ø9	- ORA - Immediate
ØA	- ASL - Accumulator
ØB	- Future Expansion
ØC	- Future Expansion
ØD	- ORA - Absolute
ØE	- ASL - Absolute
ØF	- Future Expansion
1Ø	- BPL
11	- ORA - (Indirect),Y
12	- Future Expansion
13	- Future Expansion
14	- Future Expansion
15	- ORA - Zero Page,X
16	- ASL - Zero Page,X
17	- Future Expansion
18	- CLC
19	- ORA - Absolute,Y
1A	- Future Expansion
1B	- Future Expansion
1C	- Future Expansion
1D	- ORA - Absolute,X
1E	- ASL - Absolute,X
1F	- Future Expansion
2Ø	- JSR
21	- AND - (Indirect,X)
22	- Future Expansion
23	- Future Expansion
24	- BIT - Zero Page
25	- AND - Zero Page
26	- ROL - Zero Page
27	- Future Expansion
28	- PLP
29	- AND - Immediate
2A	- ROL - Accumulator
2B	- Future Expansion
2C	- BIT - Absolute
2D	- AND - Absolute
2E	- ROL - Absolute
2F	- Future Expansion
3Ø	- BMI
31	- AND - (Indirect),Y
32	- Future Expansion
33	- Future Expansion
34	- Future Expansion
35	- AND - Zero Page,X
36	- ROL - Zero Page,X
37	- Future Expansion
38	- SEC
39	- AND - Absolute,Y
3A	- Future Expansion
3B	- Future Expansion
3C	- Future Expansion
3D	- AND - Absolute,X
3E	- ROL - Absolute,X
3F	- Future Expansion

APPENDIX B 6502 Instructions Sorted by Op-Code (Hexadecimal) *Continued*

4Ø - RTI	6Ø - RTS
41 - EOR - (Indirect,X)	61 - ADC - (Indirect,X)
42 - Future Expansion	62 - Future Expansion
43 - Future Expansion	63 - Future Expansion
44 - Future Expansion	64 - Future Expansion
45 - EOR - Zero Page	65 - ADC - Zero Page
46 - LSR - Zero Page	66 - ROR - Zero Page
47 - Future Expansion	67 - Future Expansion
48 - PHA	68 - PLA
49 - EOR - Immediate	69 - ADC - Immediate
4A - LSR - Accumulator	6A - ROR - Accumulator
4B - Future Expansion	6B - Future Expansion
4C - JMP - Absolute	6C - JMP - Indirect
4D - EOR - Absolute	6D - ADC - Absolute
4E - LSR - Absolute	6E - ROR - Absolute
4F - Future Expansion	6F - Future Expansion
5Ø - BVC	7Ø - BVS
51 - EOR - (Indirect),Y	71 - ADC - (Indirect),Y
52 - Future Expansion	72 - Future Expansion
53 - Future Expansion	73 - Future Expansion
54 - Future Expansion	74 - Future Expansion
55 - EOR - Zero Page,X	75 - ADC - Zero Page,X
56 - LSR - Zero Page,X	76 - ROR - Zero Page,X
57 - Future Expansion	77 - Future Expansion
58 - CLI	78 - SEI
59 - EOR - Absolute,Y	79 - ADC - Absolute,Y
5A - Future Expansion	7A - Future Expansion
5B - Future Expansion	7B - Future Expansion
5C - Future Expansion	7C - Future Expansion
5D - EOR - Absolute,X	7D - ADC - Absolute,X
5E - LSR - Absolute,X	7E - ROR - Absolute,X
5F - Future Expansion	7F - Future Expansion

APPENDIX B 6502 Instructions Sorted by Op-Code (Hexadecimal) *Continued*

8Ø - Future Expansion
81 - STA - (Indirect,X)
82 - Future Expansion
83 - Future Expansion
84 - STY - Zero Page
85 - STA - Zero Page
86 - STX - Zero Page
87 - Future Expansion
88 - DEY
89 - Future Expansion
8A - TXA
8B - Future Expansion
8C - STY - Absolute
8D - STA - Absolute
8E - STX - Absolute
8F - Future Expansion
9Ø - BCC
91 - STA - (Indirect),Y
92 - Future Expansion
93 - Future Expansion
94 - STY - Zero Page,X
95 - STA - Zero Page,X
96 - STX - Zero Page,Y
97 - Future Expansion
98 - TYA
99 - STA - Absolute,Y
9A - TXS
9B - Future Expansion
9C - Future Expansion
9D - STA - Absolute,X
9E - Future Expansion
9F - Future Expansion

AØ - LDY - Immediate
A1 - LDA - (Indirect,X)
A2 - LDX - Immediate
A3 - Future Expansion
A4 - LDY - Zero Page
A5 - LDA - Zero Page
A6 - LDX - Zero Page
A7 - Future Expansion
A8 - TAY
A9 - LDA - Immediate
AA - TAX
AB - Future Expansion
AC - LDY - Absolute
AD - LDA - Absolute
AE - LDX - Absolute
AF - Future Expansion
BØ - BCS
B1 - LDA - (Indirect),Y
B2 - Future Expansion
B3 - Future Expansion
B4 - LDY - Zero Page,X
B5 - LDA - Zero Page,X
B6 - LDX - Zero Page,Y
B7 - Future Expansion
B8 - CLV
B9 - LDA - Absolute,Y
BA - TSX
BB - Future Expansion
BC - LDY - Absolute,X
BD - LDA - Absolute,X
BE - LDX - Absolute,Y
BF - Future Expansion

APPENDIX B 6502 Instructions Sorted by Op-Code (Hexadecimal) *Continued*

CØ - CPY - Immediate	EØ - CPX - Immediate
C1 - CMP - (Indirect,X)	E1 - SBC - (Indirect,X)
C2 - Future Expansion	E2 - Future Expansion
C3 - Future Expansion	E3 - Future Expansion
C4 - CPY - Zero Page	E4 - CPX - Zero Page
C5 - CMP - Zero Page	E5 - SBC - Zero Page
C6 - DEC - Zero Page	E6 - INC - Zero Page
C7 - Future Expansion	E7 - Future Expansion
C8 - INY	E8 - INX
C9 - CMP - Immediate	E9 - SBC - Immediate
CA - DEX	EA - NOP
CB - Future Expansion	EB - Future Expansion
CC - CPY - Absolute	EC - CPX - Absolute
CD - CMP - Absolute	ED - SBC - Absolute
CE - DEC - Absolute	EE - INC - Absolute
CF - Future Expansion	EF - Future Expansion
DØ - BNE	FØ - BEQ
D1 - CMP - (Indirect),Y	F1 - SBC - (Indirect),Y
D2 - Future Expansion	F2 - Future Expansion
D3 - Future Expansion	F3 - Future Expansion
D4 - Future Expansion	F4 - Future Expansion
D5 - CMP - Zero Page,X	F5 - SBC - Zero Page,X
D6 - DEC - Zero Page,X	F6 - INC - Zero Page,X
D7 - Future Expansion	F7 - Future Expansion
D8 - CLD	F8 - SED
D9 - CMP - Absolute,Y	F9 - SBC - Absolute,Y
DA - Future Expansion	FA - Future Expansion
DB - Future Expansion	FB - Future Expansion
DC - Future Expansion	FC - Future Expansion
DD - CMP - Absolute,X	FD - SBC - Absolute,X
DE - DEC - Absolute,X	FE - INC - Absolute,X
DF - Future Expansion	FF - Future Expansion

Index

$15.95

6502® User's Manual,

Here is the perfect reference guide and programming tool for programmers of 6502-based microcomputers. The *6502® User's Manual* contains all the information you need for assembly and machine language programming, and for performing hardware interfacing chores. Joseph Carr, also the author of the popular *Z-80™ User's Manual,* includes comparisons of micro, mini, and main frame computers, and looks at applications categories for microcomputers in general and for the more popular 6502-based machines in particular. This valuable guide covers: Introduction to Microprocessors and Microcomputers; 6502 Architecture; 6502 Pinouts; Timing and Control Signals; 6502 Addressing Modes; 6502 Status Flags; 6502 Instruction Set (General); 65xx-Family Support Chips; Device Selection and Address Decoding; Interfacing Memory to the 6502; Interfacing I/O Devices to the 6502; Interfacing Peripherals to the 6502; Interrupts; Interfacing with the *Apple II* BUS; Interfacing with the *KIM-1, AIM-65* and *SYM-1;* 6502 Instruction Set (Detail).

0 21898 70020 9

A Reston Computer Group Book
Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

Cover design by Carol Conway

0-8359-7002-7